Open Research Online

The Open University's repository of research publications
and other research outputs

Multiple Viewpoints for Tutoring Systems.

Thesis

How to cite:

Moyse, Roderick (1991). Multiple Viewpoints for Tutoring Systems. PhD thesis The Open University.

For guidance on citations see FAQs.

© 1990 The Author

https://creativecommons.org/licenses/by-nc-nd/4.0/

Version: Version of Record

Link(s) to article on publisher's website:
http://dx.doi.org/doi:10.21954/ou.ro.0001013e

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online's data policy on reuse of materials please consult the policies page.

oro.open.ac.uk

UNRESTRICTED
DX94880

Multiple Viewpoints for Tutoring Systems.

Roderick Moyse
BA., MSc.

Submitted for the degree of
Doctor of Philosophy in Cognitive Science.

Research conducted in the Centre for Information Technology in Education,
The Institute of Educational Technology, The Open University, U.K.

November 1990.

© Roderick Moyse 1990.

Author number: M7025114
Date of submission: 2 November 1990
Date of award: 17 July 1991

ProQuest Number: 27758697

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent on the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 27758697

Published by ProQuest LLC (2019). Copyright of the Dissertation is held by the Author.

All Rights Reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Multiple Viewpoints for Tutoring Systems.

Abstract.

This thesis investigates the issue of how a tutoring system, intelligent or otherwise, may be designed to utilise multiple viewpoints on the domain being tutored, and what benefits may accrue from this. The issue was relevant to earlier systems, such as WHY (Stevens et al. 1979) and STEAMER (Hollan et al. 1984).

The relevant literature is reviewed, and criteria which must be met by our implementation of viewpoints are established. Viewpoints are conceptualised as pre-defined structures which can be represented in a tutoring system with the potential to increase its effectiveness and adaptability. A formalism is proposed where inferences are drawn from a model by a range of operators. The application of this combination to problems and goals is to be described heuristically. This formulation is then related to the educational philosophy of Cognitive Apprenticeship. The formalism is tested and refined in a protocol analysis study which leads to the definition of three classes of operators.

The viewpoint structure is used to produce a detailed formulation of the domain of Prolog debugging for novices, with the goal that students should learn how different bugs may be localised using different viewpoints. Three models of execution are defined, based on those described by Bundy et al. (1985). These are mapped onto a restricted catalogue of bugs by specifying a number of conventions which produce a simplified and consistent domain suited to the needs of novices.

VIPER, a tutoring system which can itself accomplish and explain the relevant domain tasks, is described. VIPER is based on a meta-interpreter which produces detailed execution histories which are then analysed. An evaluation of VIPER is reported, with generally favourable results.

VIPER is discussed in relation to the research goals, the usefulness of Cognitive Apprenticeship in supporting such a design, and possible future work. This discussion exemplifies the use of established student modeling techniques, the implementation of other viewpoints on Prolog, and the application of the design strategy to other domains. Claims are made in relation to the formulation of viewpoints, the architecture of VIPER, and the relevance of Cognitive Apprenticeship to the use of multiple viewpoints.

This work is dedicated to my
mother, Mite, and to the memory
of my father, Edwin.

Acknowledgements

First thanks must go to my supervisors, Dr. Mark Elsom-Cook and Dr. Diana Laurillard for their tireless help, criticism, and encouragement, and for their determination to turn a rough idea into a thesis. The prompt reading and return of draft chapters was particularly appreciated.

Thanks to Tim O'Shea for keeping an eye on things, and for the coaching on grantsmanship. Working in the Centre for Information Technology in Education has been a lot of fun and I would like to thank all the members of the group for making the three years so enjoyable. Those who waded through drafts for me must get a special mention: Pat Fung, Ann Blandford, and Fiona Spensley.

My research has benefited from discussions with many people at the Open University, and I would like to thank in particular, Claire O'Malley, Mike Brayshaw, Rick Evertz, Mike Baker and Tony Hasemer for taking the time to talk things through. Special thanks to Mike Brayshaw for always being there when I got stuck with my Prolog, which was often enough.

The wolf was kept from the door by those who helped me with consultancies and research work: Diana Laurillard, John Naughton, and Mark Elsom-Cook. In particular, thanks to John for his generosity and support, to Diana for insisting that I was worth more than was on offer, and to Mark for introducing me to so many other researchers in the field. The experience gained in this work was invaluable, and contributed much to my thesis. In relation to this I must also thank Olwyn Wilson, who always found a way to sort things out and who helped me raise the money to get to Tokyo, and all the secretarial and support staff in IET.

Many floors were slept on. A big thankyou to Claire O'Malley, Mark, Ches Lincoln and Roshni Devi for generosity beyond the call of duty.

As well as work there was music. A big hand to the guys in Virtual People for getting it on: John Gigg, Mark Elsom-Cook, and Dave Wakeley. Also a big hand to Mike Lowndes for help with booking the rehearsal rooms and gear.

A fair part of the later work was done in Edinburgh. Many thanks to everyone there for their interest, deskspace and cooperation. In particular, thanks to Paul Brna, Helen Pain and Alan Bundy for thrashing out the idea, and to Peter Ross for saying that he would buy it. Big thanks to Judy for printout, to Jean Bunton and Millie Tupman who saved me from taking my machine home every night, and to Karen for oiling the wheels.

Acknowledgements

Many others, here and abroad, have had an influence on my work. I would like to thank Allan Collins, Gordon McCalla, Joost Breuker, Radboud Winkels, Pierre Dillenbourg, Benedict du Boulay, Jim Greer, Peter Goodyear, John Self and Rachel Rimmershaw for their interest and encouragement. Special thanks to Peter for the invitation to Calgary and for bracing walks.

Most of all, thanks to Karen for making it all worthwhile.

This research was supported by a postgraduate award from the Science and Engineering Research Council of the United Kingdom.

Table of Contents.

Contents.....i
Table of Figures.....vii
Table of Tables.....viii
Publications.....ix
Chapter 1. Introduction to the Thesis.....1
1.1 Viewpoint Structures and Related Issues.....1
1.2 Designing the System and Formulating the Domain: the influence of the Viewpoints.....9
1.3 The Domain Formulations and the Dialogues they Support.....14
1.4 Design Considerations and Claims for VIPER.....21
1.5 Conclusions.....25
1.6 The structure of the thesis.....26
Chapter 2. Viewpoints in tutoring systems: uses, structures, and domains.....29
2.1 Viewpoints in ITS design: the problem.....29
2.1.1 Different viewpoints required for the same activity.....30
2.1.2 Different Viewpoints for different activities.....34
2.1.3 Different viewpoints from novice to expert.....35
2.1.4 Glass boxes and black boxes.....36
2.1.5 Exploring the student's viewpoint.....38
2.1.6 Viewpoints and design philosophies.....39
2.1.7 Conclusions: defining the problem.....44
2.2 Mental Models.....47
2.2.1 Studies of mental models.....48
2.2.2 The application of models.....51
2.2.3 Conclusions.....54
2.3 Prolog Tutoring and Tracing.....54
2.3.1 Prolog novices have problems.....55
2.3.2 A technique for interpreting a record of Prolog execution.....57
2.3.3 Conclusions: Prolog as an implementation domain.....58
2.4 Debugging: studies and tutoring.....58
2.4.1 Describing Bugs.....59
2.4.2 Debugging Systems.....60
2.4.3 Conclusions: Prolog debugging as an implementation domain.....62
2.5 Explanation Content and Knowledge Base Structure.....63
2.5.1 Structuring the Knowledge Base.....63
2.5.2 Interpreting the Knowledge Base.....66
2.5.3 Conclusions.....69
2.6 Educational Philosophy.....70
2.6.1 Ways of learning.....70
2.6.2 Applying Knowledge.....76
2.7 Conclusions to Chapter 2.....77

Contents.

Chapter 3. A Formulation for Viewpoints.....80
3.1 Considerations for the formulation.....80
3.2 The goals of the formulation.....86
3.3 The Formulation.....88
3.3.1 The structure for implementing viewpoints.....88
3.3.2 The viewpoint structure and "cognitive apprenticeship".....93
3.3.3 The viewpoint structure and alternative possible structures.....96
3.4 Extending and testing the proposed viewpoint structure.....98
3.5 Conclusions to Chapter 3.....99
Chapter 4. Testing the Formulation: A Protocol Analysis.....100
4.1 Introduction.....100
4.1.1 Goals of the study.....100
4.1.2 Outline of the study method.....101
4.2 Details of the study method.....102
4.2.1 Models and Instructions.....102
4.2.2 Training Systems.....106
4.2.3 The Simulation Screen.....107
4.2.4 The Simulation Algorithms.....107
4.2.5 The Subjects.....109
4.2.6 Expected results and purpose of the study.....109
4.3 The Protocol Analysis.....110
4.3.1 The differential analysis.....110
4.3.2 Segmentation and Validation.....111
4.3.3 Protocol Analysis Results.....112
4.4 Discussion of the Protocol Analysis Results.....113
4.4.1 Introduction.....113
4.4.2 The Steamvalve Control.....114
4.4.3 The Damping Control.....116
4.4.4 The Pumpspeed Control.....118
4.4.5 Errors in the reasoning.....119
4.4.6 Responses to extreme system states.....119
4.4.7 Problems involving the simulated system.....121
4.5 Conclusions to Chapter 4.....122
Chapter 5. Testing the Formulation: Formalising the results of the Protocol Analysis.....124
5.1 Introduction: The Goals of the Formalisation.....124
5.2 Formalising the reasoning of the subjects.....126
5.2.1 A Formalisation of the models.....126
5.2.2 General remarks concerning the formalisation.....128
5.2.3 Formalising the reasoning of the 'functional model' group.....129
5.2.4 Formalising the reasoning of the 'structural model' group.....131
5.2.5 Conclusions: formalising the reasoning of the two groups.....134
5.3 Classes of operators required to formalise the reasoning.....134

5.3.1 Introduction. 134
5.3.2 Operator type one: the 'access' operator....135
5.3.3 Operator type two: the 'inference' operator....135
5.3.4 Operator Type Three....136
5.4 Conclusions to Chapter 5....137
Chapter 6. The Implementation Domain and Tutoring Goals....138
6.1 Introduction. 138
6.2 Prolog for novices....142
6.2.1 Introduction. 142
6.2.2 Background to the models of Prolog execution....143
6.2.3 Prolog Model 1: The Database of Clauses....145
6.2.4 Prolog Model 2: The Search Space....146
6.2.5 Prolog Model 3: The Search Strategy....148
6.2.6 Prolog Model 4: The Resolution Process....151
6.2.7 The models combined as an interpreter. 152
6.2.8 Conclusions to section 6.2 153
6.3 Describing Bugs 154
6.3.1 Classifying Bugs: a simplified environment....154
6.3.2 The Bug Trees and their application....159
6.3.3 Conclusions to section 6.3. 163
6.4 Formulating abstract descriptions of the effect of each bug....163
6.4.1 Introduction. 163
6.4.2 Bug effects on Program execution. 164
6.4.3 Developing description templates for each bug execution pattern....171
Introduction....171
Conventions governing the mapping of models onto bugs. 171
Classifying bugs in terms of the models....176
Questions which may be asked of the student....178
Building templates to describe bugged execution....180
Template substitution. 184
6.4.4 Conclusions to section 6.4. 185
6.5 Re-formulating the Models of Prolog for tutorial dialogues....186
6.5.1 The dialogues required....186
6.5.2 'Procedural' versions of the Prolog models. 188
6.6 Conclusions to Chapter 6....190
Chapter 7. VIPER: Viewpoint-based Instruction for Prolog Error Recognition. 193
7.1 Introduction. 193
7.2 The Meta-interpreter. 197
7.2.1 The meta-interpreter: top level....197
7.2.2 Implementing the Resolution model....199
7.2.3 Implementing the Search Strategy model....200
7.2.4 Implementing the Search Space model. 205

7.3 Recording and analysing the execution history. 206
7.3.1 Asserting the execution history facts 206
7.3.2 Analysing the execution history facts 207
7.3.3 Data available after execution history analysis. 209
7.4 The tutorial dialogues. 210
7.4.1 Introduction. 210
7.4.2 Using the procedural versions of the Prolog models. 210
Dialogue 1: describing execution. 211
Dialogue 2: Identifying the bugged clause. 220
Dialogue 3: Describing the bug's effect on execution 226
7.5 Conclusions to Chapter 7. 231
Chapter 8. Evaluating VIPER. 233
8.1 Introduction. 233
8.2 Method and Materials 235
8.2.1 A tutorial with VIPER 235
8.2.2 Printed Materials 237
8.2.3 The Questionnaire. 238
8.3 Results. 239
8.3.1 Responses to Questionnaire section 1, experience of Prolog. 239
8.3.2 Responses to Questionnaire section 2, the Interface. 240
8.3.3 Responses to Questionnaire section 3, the Viewpoint Representations. 244
8.3.4 Responses to Questionnaire section 4, VIPER overall 249
8.3.5 Responses to Questionnaire section 5, the Viewpoints. 254
8.4 Discussion 255
8.4.1 Discussion of responses to Questionnaire Section 1 255
8.4.2 Discussion of responses to Questionnaire Section 2 256
8.4.3 Discussion of responses to Questionnaire Section 3 258
8.4.4 Discussion of responses to Questionnaire Section 4 260
8.4.5 Discussion of responses to Questionnaire Section 5 264
8.5 Conclusions to Chapter 8. 264
Chapter 9. A Discussion of VIPER 266
Introduction 266
9.1 The viewpoint formalisation. 266
9.2 Design goals of VIPER 269
9.3 VIPER's design. 271
9.3.1 VIPER and Debugging 271
9.3.2 Black boxes and Glass boxes. 276
9.3.3 Successful and unsuccessful aspects of VIPER's dialogues. 277
9.3.4 System architecture. 278
9.4 Cognitive Apprenticeship. 279
9.5 VIPER and other domains 281
9.5.1 Introduction. 281

9.5.2 Another viewpoint on Prolog. 282
The viewpoints which utilise the process history. 282
The production of the process history. 284
9.5.3 A different domain. 284
The viewpoints which utilise the process history. 285
The production of the process history. 287
9.5.4 A domain that cannot be implemented using VIPER's strategies. 292
9.6 Future Work. 292
9.6.1 Introduction. 292
9.6.2 Domain Models and Domain. 293
9.6.3 System Architecture. 296
Diagnosis. 296
9.6.4 Dialogue 1: Execution Description. 298
Diagnosis 299
9.6.5 Dialogue 2: Identifying the bugged clause. 301
Diagnosis. 302
9.6.6 Dialogue 3: Describing the bug's effects. 302
Diagnosis. 303
9.7 Conclusions to Chapter 9. 306
Chapter 10. Conclusions 310
10.1 Research Goals. 310
10.1.1 The viewpoint formalism 310
10.1.2 The system architecture. 312
10.1.3 Cognitive Apprenticeship. 313
10.2 Pedagogical goals served by the use of multiple viewpoints 314
10.2.1 Adaptation to the goals of the student 314
10.2.2 Explanation in terms of different viewpoints 315
10.2.3 Sub-dividing the domain. 315
10.2.4 The elimination of misconceptions. 315
10.3 Summary of Claims. 317
References. 318
Appendix 1. An example execution history from VIPER I
Appendix 2. The documents used for VIPER's evaluation. II
Appendix 2.1: VIPER evaluation briefing document. II
Appendix 2.2: Models used in VIPER's evaluation for the description of a subset of Prolog Execution. IV
Appendix 2.3: The menu options relating to each model. V
Appendix 2.4: The queries and program databases used in Dialogue 1. VI
Appendix 2.5 The possible bugs associated with each viewpoint. VII
Appendix 3. The queries and code used for Dialogues 2 and 3 of the evaluation. VIII
Resolution Viewpoint VIII
Search Strategy Viewpoint IX

Contents.

Search Space Viewpoint..... X
Appendix 4. The Evaluation Questionnaire..... XI

Table of Figures.

Figure 1: An Outline Structure for Viewpoints 89

Figure 2: The simulation screen in starting state 103

Figure 3. The number of encodings of the specified category in the first twenty minutes of each session for each pair of subjects. Grey columns represent the pairs given a structural model. Patterned columns represent the pairs given a functional model 113

Figure 4. An overall view of the work described in chapter 4 141

Figure 5. The Bug Tree for the symptom 'Unexpected instantiation of a variable'. (Symptom from Brna et al. 1987) 158

Figure 6. A summary of the relationships of Symptom, Bug, Module, Models of execution, and Templates. 161

Figure 7. The 'Bug Tree' for the symptom 'A variable instantiated to an unexpected value'. (Symptom from Brna et al. 1987) 162

Figure 8. The description template derivation tree for the symptom 'A variable instantiated to an unexpected value' combined with a specific bug, bug manifestation and question. (Symptom from Brna et al. 1987) 180

Figure 9. The structure of VIPER 195

Figure 10. A partial call graph for the meta-interpreter: 'order_res_truth' checks a goal against successive database clauses 198

Figure 11. An example of the explanations used in Dialogue 1 212

Figure 12. The explanation of figure 11 applied to the current execution. 213

Figure 13. The procedural Search Space model with its associated symbols, explanation templates and menu versions of model parts. 217

Figure 14. The screen for describing code execution 219

Figure 15. The screen for describing code execution showing the Resolution model menu 'popped-up' to take student input 219

Figure 16. The screen for identifying the bugged clause 221

Figure 17. The screen for describing the bugged execution 228

Table of Tables.

Table 1. Possible legal goal outcomes at a given point in the traces of both Ideal and Bugged code for the bug 'Missing Clause' and the symptom 'A variable instantiated to an unexpected value', using models of section 4.2 and database conditions of section 4.3. (Symptom from Brna et al. 1987). 168

Table 2. Possible legal goal outcomes at a given point in the traces of both Ideal and Bugged code for all bugs of section 4.3. and the symptom 'A variable instantiated to an unexpected value', using the models of section 4.2 and database conditions of section 4.3. (Symptom from Brna et al. 1987). 169

Table 3. Each row in the table gives a recognised Code Error, the model associated with it, and the relevant comment on the Search Space. (Code Errors derived from Brna et al. 1987) 178

Table 4. Alternative slot entries for Functor, Arity, Argument, and Search Strategy slots of templates to describe execution of bugged clause and relevant goal. Rows are not meaningful..... 183

Table 5. Complete template to describe the execution and analysis of version 1 of the bug 'Extra Clause' when the relevant clause has no subgoals. 'Version 1' implies that the traces of both bugged and ideal databases show a successful resolution at the same point. 184

Table 6. The symbol, rule and explanation template combination for the execution event where the functors of a goal and clause unify. 214

Table 7. The proceduralised models abbreviated to a set of menu choices, with their associated system symbols. 215

Table 7 Continued. 216

Table 8. Options available on the 'Questions' menu. 224

Table 9. The input options available for each button of the 'Bugged Execution' dialogue. 229

Table 10. Examples of possible execution history segments for a proposed 'Heavy Rainfall' tutor.... 288

Table 10 Continued..... 289

Publications.

Parts of the work described in thesis have been published as follows:

1. The contents of chapter 3, (the protocol analysis study), with the exception of the operator formalisations, was published as :

> Moyse R. (1991). "Multiple Viewpoints Imply Knowledge Negotiation" in Interactive Learning International, vol. 7 part 1. (January '91).

2. An outline of the domain formulations and implementation described in chapters 4 and 5 was published as:

> Moyse R. (1990). "Implementing Knowledge Negotiation". Proceedings of the International Conference on Advanced Research on Computers in Education, Tokyo, Japan, July 18 - 20th. 1990. North-Holland.

3. An early summary of the viewpoint formulation of chapter 2 and the protocol analysis study of chapter 3 was published as:

> Moyse R. (1989c). "Knowledge Negotiation Implies Multiple Viewpoints", in Artificial Intelligence and Education: Proceedings of 4th. International Conference on AI and Education, Amsterdam, May 1989. .IOS, Amsterdam.

4. An overview of VIPER's design and evaluation is now in press as:

> Moyse R. (in Press). "Design Strategies for Knowledge Negotiation: The VIPER system", in Moyse R. and Elsom-Cook M.T. (Eds.) Knowledge Negotiation, Paul Chapman, London.

5. A short paper summarising the contents of chapters 1 and 2 was published as:

> Moyse R. (1989b). "Multiple Viewpoints for Intelligent Tutoring Systems", Interactive Learning International, vol. 5, no. 2. April 1989.

Chapter 1. Introduction to the Thesis.

1.1 Viewpoint Structures and Related Issues.

The literature of Intelligent Tutoring Systems (ITS) contains many references to the virtues of using multiple representations of the domain knowledge, the purpose being to express different viewpoints upon that domain. A standard example is the WHY system described by Stevens, Collins and Goldin (1979). They concluded that to tutor the causes of heavy rainfall properly it was necessary to represent 'functional' knowledge of particular processes, such as condensation, as well as 'scriptal' knowledge ie. knowledge about the sequence of conditions which leads to the downpour. Other examples include QUEST (White and Frederiksen 1986) which highlighted the differences between experts' and novices' views of electrical circuits, and the research conducted for STEAMER (Hollan, Hutchins, and Weitzman 1984) which showed that the way experts think about a particular machine does not necessarily depend on its physical components but on their intended actions. This thesis examines these and other examples and identifies a range of issues as being involved in the use of multiple viewpoints for tutoring systems.

The issues identified are:

- The need to use multiple viewpoints to correctly carry out a single activity;
- The need to use multiple viewpoints to carry out different activities;
- The need to employ different viewpoints at different points along a learning path;
- The possibly contrasting needs for system efficiency in task execution, and for clarity in explanations given to the student;
- The desire to promote reflection and meta-cognition in the student;
- The desire to explore the student's own viewpoint;
- The need to formulate an adaptive and constructive philosophy of tutoring system design, which does not rely on a "transmission" theory of education (ie. a theory where knowledge is viewed as something that has to be simply moved from system to student).

These issues, especially the last issue listed above, indicate that the use of multiple viewpoints is of profound and general importance in the design of tutoring systems. This thesis develops a design methodology related to their use and indicates that the issues should be considered as central to every stage of system design. The importance of the issues can be briefly illustrated by reference to the work of Self (1988a, 1988b) who argues that reliance on a single representation of the tutored domain has led to an inappropriate 'trinity' model of system design; (the student model, the domain representation, and the tutoring component). If this model leads designers to assume that the system has, in a Platonic sense, the 'one true' representation of the domain then they may also assume that the student may be adequately represented as a 'subset' of the expert, or domain representation. This can easily lead to an authoritarian style of tutoring with an implicit 'transmission' theory of education being applied.

In this context Self advocates the use of belief systems to represent domains and the various viewpoints upon them. To this end Self (1990) has furnished us with a wide-ranging review of systems which attempt to model the holding and revision of sets of beliefs. Unfortunately this review indicates that the construction and processing of belief systems is an area beset by fundamental technical difficulties. This being so the issues surrounding their use in tutoring systems must be equally unclear. The direction taken in this thesis is to re-define the problem, and to investigate what can be achieved with *pre-defined* viewpoints. What is thus required is a methodology for structuring viewpoints and for designing systems which utilise them, which takes account of all the relevant issues. Specifically this means that the domain representations of the system being designed must be formulated in terms of the viewpoints that we wish to use and the way we wish to use them, and that the tutoring interactions that the system is designed to support must be oriented to the availability of multiple viewpoints on the domain.

The design methodology described in this thesis relates these factors and specifies the particular kinds of adaptation to the student that can be achieved by the use of pre-defined

viewpoints. The first step is to develop a single structure which can be used to represent various viewpoints on various domains. Once this has been done an example domain is chosen and an example system is implemented to investigate the kinds of adaptability that can be achieved. The results indicate that if the viewpoints are carefully chosen tutoring may be adapted to a student's goals, to their previous experience of the chosen and other domains, and to any misconceptions that they may be harbouring. Also, since the structure for formulating viewpoints emphasises the *use* or *application* of the knowledge being learned, tutoring may be conducted in terms of this meta-knowledge. The paths of influence in the design process are thus circular or iterative: the viewpoints chosen for implementation in the system will depend on the kinds of goal, experience and misconception that are envisaged as being present in the target student population and on the kind of tutoring that the designer wishes the system to carry out. At the same time, the kinds of tutoring dialogue and adaptation that are possible will depend on the specific viewpoints that can be represented and on the interactions that they will support. To the extent that viewpoints are chosen and structured the design method *attributes* a structure to the domain. To the extent that the tutoring interaction is adapted to the viewpoints that the domain affords the design method *exploits* the structures that are inherent in the domain. These two possible influences must be reconciled to produce an effective system.

The viewpoints referred to in this thesis are more than just alternative representations. This may be shown by representing the same information (eg. the novice's view of an electrical circuit) in two different formalisms. The information content would be the same in each case although the representations of that information would be different. What then, does "viewpoint" mean? We shall try to answer this firstly by exploring a specific example and then by proposing a general structure for describing viewpoints.

The example is from Minsky (1981) who points out that for hard problems one "problem space" (ie. an initial state, a goal state, and a set of operators) is not usually enough. Troubleshooting the electrical system of a car for example, may require that we use either

an electrical or a mechanical viewpoint. Each has its own set of labels for the same set of objects (eg. the car body as earth connection) and its own set of diagnostic questions. While electrical faults generally require mechanical manipulation in order to rectify them, the isolation of the fault may well require that both modes of analysis be used separately. This implies that there is some other form of knowledge involved, a 'control knowledge' which allows decisions to be made about when and how each mode of analysis is to be used.

The viewpoints described in the previous paragraph may be characterised in two ways. Firstly, they are complementary modes of analysis both of which may be required in a particular context. Secondly, they refer to the same set of objects but identify and structure them in quite different ways. The value of this seems to be that each view brings out particular features of the domain in question.

With these points in mind the concept of a "viewpoint" is proposed as a theoretical construct which is useful in ITS design. It is not proposed as a psychological reality. Viewpoints are initially conceptualised as being similar to the "...user's conceptual model..." described by Young (1983). While acknowledging a diversity of definitions for such models, Young sought to summarise a general agreement about their nature by describing them as:

> "...a more or less definite representation or metaphor that a user adopts to guide his actions and help him interpret the device's behaviour."

In order to describe viewpoints these models need to be augmented with a range of operators which can be used to infer different information from the model, or else transform it. This need can be illustrated by considering a model which describes some parts of a car alternator system and their relationships. (This example is explored in detail in chapter 2). Where the model describes a rotor, pulley wheels, wires and drive-belts and their correct relationships it could be used to answer the two distinct questions a) "does the

system have a drive-belt?", and b) "Is this particular drive-belt in the correct condition?". Each answer is obtained by performing a different operation on the model.

These inferential operators are seen as vital to the proper definition of a viewpoint for implementation. One reason for this lies in the belief that in order to tutor and explain a domain effectively a tutoring system must itself be able to perform the tutored tasks in the given domain. It must thus be able to draw the required inferences from a given model and so must be equipped with suitable means of doing so. Also, as the Minsky example above indicates, we need some 'control knowledge' to decide when a particular model is applicable to a particular problem. Moyse (1989) suggests that this may be achieved by equipping each viewpoint with heuristics which describe the kinds of problems to which it may be applied. As mental models have to be applied to problems, and as we wish to describe this process, a viewpoint is seen as a description of the application of a mental model.

The structure for viewpoints was developed with the intention that it should be used as a basis for formalising viewpoints which could then be implemented as the domain representations of an ITS. Before starting the implementation, it was thought wise to actually test the structure's ability to formalise different viewpoints on a domain, and develop it as necessary.

This testing and development required that some methodology be adopted. Since no text-book was available to specify an appropriate methodology, methods used elsewhere in cognitive science were harnessed to a new purpose. The methods had been used to investigate the application of mental models to specific systems but had not studied the specific inference processes used with each model. Since these inference processes were seen as a vital part of each viewpoint, a study was designed which enabled us to record the decision-making process as verbal protocols. The protocols were then analysed to determine which inference processes were used. The protocols were obtained by asking

groups of subjects to apply different models to the same simulated system with the same specific goals, and recording the resulting dialogues.

The purpose of this exercise was twofold:

- to determine whether the intended structure for viewpoints was adequate. If the models used by different groups and the inference procedures associated with each model could be shown to be quite distinct, and if the intended structure for viewpoints could be used to formalise *both* combinations of model-and-set-of-inference-procedures, then it could be concluded that the intended structure was sufficiently robust to serve as the basis of an implementation.
- to determine what kinds of operator would be required. The notion of an 'operator' is a very general one. The implementation of a system requires that specific operations be defined. In order to fulfil our goals for the tutoring system design we needed these operations to reflect, as far as possible, the reasoning used by human users. It was assumed that the analysis and formalisation of reasoning exhibited by human subjects could help to define classes of operator which satisfied this need.

The study is described in Moyse (1991). It resulted in an extension and clarification of the proposed structure for implementing viewpoints. Two groups of subjects were asked to control a simulated power station, having been given different models of the simulation. One group were told, broadly, that it was a "power generation system" and were given a functional model of the system which simply linked specific actions to specific effects, while the other group were told that it was a "nuclear power station", and were given a structural model. (Young [1983] describes structural models as "surrogate" models which allow a user to make inferences about a system's behaviour). The decision-making conversations of each pair were recorded and transcribed as verbal protocols. These were analysed to reveal what kinds of inference were involved with each model.

One purpose of the exercise was to show that the proposed structure for viewpoints was useful and effective in modelling the different inference processes which might be involved with the use of different mental models. Under these conditions, the structural model appeared to entail a much greater use of real-world knowledge and causal reasoning. The functional model tended to elicit rule-based, or condition-action based reasoning. These processes were formalised in terms of the proposed structure for viewpoints. The fact that the viewpoint structure was able to encode the quite distinct mental models and reasoning processes of the two groups was taken as confirmation of its usefulness as a means of formalising different viewpoints. The study also allowed us to develop descriptions of three categories of operators which could be used to draw inferences from the models. The use of heuristics to encode information detailing the *use* of each viewpoint was not tested in this study. We do not wish to make strong claims as to the precise psychological mechanism (eg. analogy, qualitative reasoning) involved in the manipulation of the models described, but suggest that the notion of a viewpoint could encompass any or all of these mechanisms.

Having verified the usefulness of the viewpoint structure it was necessary to choose an example domain in which to implement a tutoring system. A very strong motivation at this point was that a *real* domain should be chosen. By this we mean a domain where the actual viewpoints used by practitioners or students in that domain could be formalised and used and some attempt made to deal with the real problems and misconceptions that they faced. There were several reasons for this motivation. The first and most general was a belief that the appropriate way for tutoring systems technology to advance was for it to engage with real rather than 'toy' domains, and their related problems. The second reason related to the intended evaluation of the implemented system. If a fictional domain, or an excessively arcane one were to be chosen, then serious problems could arise in finding a sufficiently large user population to conduct some form of evaluation. A third reason was the simple wish that the work done should at least have the chance of being of some practical use when it was completed. It was felt that the issues involved were of

considerable significance to system design, and to education generally, and that every attempt should be made to demonstrate this to those who were not yet convinced.

These considerations meant that the domain used for the study described above (nuclear reactor/power station) was not suitable. True, there is a pressing need for reliable and effective training to be given to the operators of such systems in the real world, but the nature of these systems and the complexity of the knowledge required to deal with them meant that the accurate acquisition and representation of such a domain was a project which exceeded the resources available for completion of the thesis. It was also foreseen that there would be considerable problems in finding a suitable pool of practitioners who could take part in an evaluation.

What was required was a 'real' domain of suitable size where the relevant viewpoints had at least been identified, where a definite educational need had been established, and where a sufficient number of users could be found to conduct an evaluation after the system had been implemented. These needs were answered by the domain of Prolog, especially of Prolog for novices. The language is frequently a mystery to those who are encountering it for the first time, and sometimes even to those who are more experienced in its use. Attempts to alleviate this situation have included the description of a series of models (Bundy et al. 1985) which may be taught to novices so as to give them a more structured initial understanding of the language. We assumed that viewpoints based on these models could be developed. As the structure used to define viewpoints emphasises the use of the knowledge involved, we wished to define a domain where the different viewpoints could actually be applied, and where such application could be practised and critiqued as a part of the tutoring process. Models of execution are clearly necessary in the task of debugging. If students have problems understanding how Prolog works, then they will have even greater problems in debugging code. These considerations led to the choice of Prolog debugging for novices as the experimental application domain. The goal was to build a

system which could tutor the skill of using different viewpoints to localise bugs in Prolog code.

1.2 Designing the System and Formulating the Domain: the influence of the Viewpoints.

The first step in the design of the system which could tutor the use different viewpoints to localise bugs in Prolog code (VIPER: Viewpoint Based Instruction for Prolog Error Recognition) was thus the definition of the necessary viewpoints on Prolog. Models were required to act as the core of each viewpoint. An initial set of models was defined based on those described in Bundy et al. (1985). Bundy et al. outline four complementary "user's conceptual models" (Young 1983) of a Prolog interpreter which can be taught to novice students to help them understand Prolog execution. These are the Program Database, the Search Space, the Search Strategy, and the Resolution Process. In various combinations, these four models can be used to comprehend the many different representations of Prolog execution such as Byrd Boxes, Arrow Diagrams, and And/Or Trees.

Three of the four models from Bundy et al. were re-defined and changed somewhat, formalised, and implemented (incidentally in Prolog code). The fourth model was not used because the bugs it could localise were mainly concerned with syntax errors. It was assumed that syntax errors would be caught by the host environment. The remaining three models were changed to suit the goals of the design and the needs of the domain formulation.

When the three models are combined via a control structure, they constitute an interpreter which exhibits a subset of Prolog behaviour, and which can be used to produce an execution history in the required terms. This 'history' can then be interpreted. This ability to 'watch' a 're-play' of the interpreter's execution allows the tutoring system to describe that execution in terms of the three models outlined above. The purpose of this is to tutor a set of viewpoints on (a subset of) Prolog execution which novice programmers may use to

localise bugs in their code. These initial models are intended to be the first in an upwardly-compatible progression, and as such do not cater for backtracking or the use of the 'cut'. These aspects of Prolog are not seen as crucial to our preliminary goals for the prototype system, as we only intend to use problems which do not depend on backtracking for their solution. Our intention is that more sophisticated execution models which describe backtracking should be developed when the initial system design has been proven.

To be fully implemented as viewpoints, the models need to be associated with a set of inference operators of the kind described in Moyse (1989, 1991) and an indication of each viewpoint's area of application. The models presented to the student are cast in the form of a set of procedural 'if-then' rules to facilitate their use in the description of execution. The 'if' part specifies when a particular rule should be applied, and the 'then' part specifies what execution step should occur.

The structure of the resulting viewpoints was central to VIPER's design. In terms of the design method outlined above the viewpoints were chosen or "attributed" to the domain of Prolog debugging, as they facilitated the tutoring of novices, and gave an account of Prolog execution which was 'complete' in the sense that novices could use them to perform all the tasks with which they would be presented in the tutoring interaction. The viewpoints "exploited" structures present in the domain of Prolog debugging in the sense that they used such concepts as "Search Space" and "Search Strategy" which are widely used by Prolog practitioners, and which can be used (or frequently are used) to define sets of possible bugs. The point of this attempt to 'exploit' the domain structure is that if the system can perform the domain tasks in terms which are at least related to those used by human practitioners then it should be able to tutor, demonstrate, and explain those tasks in the same terms.

When combined with the requisite operators, the models constitute the viewpoints which are the first part of the domain formulation of VIPER. VIPER performs tasks in the

domain by using the inference operators to act on the models of Prolog execution. The tutoring tactics and strategies adopted are based, in part, on the possibilities afforded by these domain formulations. (It should also be remembered that the formulations were chosen partly because of the tutoring strategies that they made possible). In the actual implementation the heuristics which define the area of application for a viewpoint are not explicitly implemented. Instead, the possible bugs are defined in such a way that a given bug can *only* be related to a single viewpoint. (This is described in more detail in section 1.3). This means that the area of application of a given viewpoint (ie. its use to localise specific categories of bug) is made obvious as it is one of the core structures or conventions of the domain. The structure that was adopted for implementing viewpoints thus has a direct influence on the way that the VIPER's domain is formulated.

It is the relationship between three specific viewpoints and the categories of bugs that they can localise which forms the second part of the domain formulation. This describes a simplified environment of bugs and 'ideal' solutions in terms drawn from Brna et al. (1987). Brna et al. give a four-level classification for bugs where the descriptions are related to programmer expectations. A simplified environment is necessary in order to avoid the serious difficulties inherent in trying to build an intelligent 'debugger' and to allow available resources to be used in investigating the tutoring aspects of the problem; (ie. the goal is to tutor about debugging without trying to build a full-blown debugger). The space of possible bugs is thus divided into three sets, each of which relates to a particular viewpoint in such a way that operators can be defined which allow the relevant viewpoint in VIPER to identify the bug. The mapping from viewpoints onto bugs is thus strongly influenced partly by the need to exploit a particular structure which is used to formalise viewpoints, and partly by the need to simplify an intractable domain.

The use of viewpoints formalised in the way described influenced other parts of VIPER's architecture. The first of these is the use of an execution history. In order to determine the nature and effect of a specific bug VIPER compares the execution of the bugged code with

that of an ideal version of the code. VIPER needs to know the results of the two executions to determine firstly whether a bug is present and, if so, what its exact nature may be. A history of each execution must thus be stored in order for VIPER to compare them. The use of viewpoints and of a specific viewpoint structure influences the *terms* in which this history is recorded. It is not sufficient to record the execution in terms of a single viewpoint. Instead the history must contain all the information required for an analysis in terms of any one of the three implemented viewpoints. The terms used to record this execution history must be recognisable by the viewpoints and their operators. This in turn implies that the interpreter which runs the code must be built so as to produce the kind of execution history which is required. The point to be made here is that the influence of particular viewpoints structured in a particular way extends to the very core of the system design.

Another feature of the architecture influenced by the viewpoint formalisation and its implementation in the specific domain chosen, was the nature of the dialogues that the system would support. The viewpoint structure emphasises the application of the knowledge involved. In this case this amounts to the localisation of a bug. In order for the system to determine that the student has done this correctly, it must have positive evidence that two conditions are satisfied. Firstly, there must be evidence that the student has correctly identified the bug. However, correct answers here could be the result of guesses or random choice so a second condition must be satisfied: there must be evidence that the student correctly understands the effects on execution implied by the chosen bug. The investigation of each condition requires a separate dialogue. The point to be made is that the choice of a viewpoint structure which emphasises the application of the knowledge involved has implications for the kinds of tutorial dialogue that are deemed to be relevant and necessary.

The emphasis on this aspect of the viewpoint structure has one other influence that should be mentioned. It influences the kinds of extension to the system and further work that are

envisaged for VIPER. Where the area of application for a viewpoint is made explicit and is available to the system a range of adaptations to the student become possible. For example a statement of the student's goals could be matched to a statement of what problems given viewpoints can be applied to and a suitable viewpoint chosen for tutoring. Alternatively, where a student is manifestly applying the wrong viewpoint to deal with a problem, (eg. a mechanical rather than an electrical view of a motor ignition problem), the missing 'viewpoint application' or 'meta-knowledge' could be taught directly.

This section attempts to describe a number of ways in which the structuring and implementation of viewpoints is intimately bound up with the central issues of tutoring system design. The structure of models, operators, and heuristics which is adopted to formalise and implement viewpoints is intended to be general, and applicable to many domains. The particular domain formulation that results from applying the structure to Prolog debugging for novices, and the specific tutorial dialogues and system structures that are required to exploit it, are specific to the chosen domain. The strategy of using models and an execution history which are analysed or interrogated by operators is seen as being applicable to many procedural domains. The precise content of this history will in each case depend on the nature of the viewpoints implemented, as will the detailed structure of the system which generates the history. The use of heuristics to describe each viewpoint's area of application is intended to be something which could be achieved in every suitable domain and would thus enable similar forms of adaptation to the user in each of those domains.

1.3 The Domain Formulations and the Dialogues they Support.

This section describes the formulated domain, and the dialogues supported by it in more detail.

The operational goals of VIPER are that the student should be presented with standard novice-level programs which contain a single bug. The student's task is to learn how to localise the bugs. The system, if it is to tutor effectively, must also be able to locate the bugs. This does not mean that VIPER constitutes an intelligent debugger. Such systems have to deal with arbitrary code structures and multiple solutions to a single problem. VIPER utilises an ideal solution to each problem set to the student and compares the execution history of this solution with the execution history of the bugged code in order to locate the bug. Our intention is not so much that the system should be able to find the bug, but that it should promote the skill of searching for it in terms of the procedural execution models. The use of an ideal version of the code allows us to concentrate on this pedagogical goal with the system setting the agenda and determining which specific bugs may be dealt with in a given tutoring episode.

VIPER deals with a restricted category of bugs which are described in terms defined by Brna et al. (1987). This allows us to systematically describe the bugs that the system can handle and thus the classes of problems that may be incorporated as tutoring materials. The allowed bugs are described in terms of missing, extra, or wrong 'modules'. Depending on the level of description 'modules' may be such things as whole predicates, subgoals, or arguments. Bugs may be concerned with termination or with variable instantiation issues. Bugs concerned with variable instantiation may give rise to a) the unexpected failure to instantiate a variable; b) the unexpected instantiation of a variable; c) a variable instantiated to an unexpected value.

The missing, extra, or wrong modules which may give rise to this behaviour may be listed. The search strategy of Prolog requires that we also add to the list the possibility of wrong order for clauses and subgoals. If our list of modules is complete then all possible (individual) bugs may be described in this way in relation to the ideal template code. Although this classification is 'syntactic' in that it does not include any of the procedural semantics of the programmer, it has the advantages of being simple, regular, and complete, while defining a finite number of bug types. The catalogue of bugs is related to the models of execution by a set of conventions which determine what is, for instance a 'Resolution' bug, and what is a 'Search Strategy' bug. This 'mapping' from models to bugs allows us to state that certain bugs can be localised by using certain viewpoints.

Using Brna et al.'s (1987) classification we may define 'trees' of possible bugs for each of the three kinds of variable symptom. The 'instantiation to an unexpected value' bug for instance implies that a goal containing variables succeeds in both the ideal and bugged code. Thus only bugs capable of yielding this result need to be considered. These 'trees' can also be used to specify the range of possible bugs which may be included in the problems set to the student if we stipulate that only a single bug may be present in each problem. For reasons of tractability and clarity, we also stipulate that the bugged code may only have one difference from the ideal code, that of the bug chosen.

The pedagogical goal of the system is that the student should develop the ability to describe Prolog execution in terms of the execution models so as to localise possible bugs. VIPER facilitates the first part of this learning by asking the student to use the models to describe the execution of a given query and set of clauses. Where necessary VIPER can demonstrate this skill. The student's description is checked against an execution trace which contains much more information than a normal trace would do. Each goal has to be shown being matched against all clauses in the program database, not just against those which share the same functor. If a given resolution fails then the precise reasons for this must be reported so that the different types of failure can be made clear.

To this end VIPER's meta-interpreter records each step of the execution as a series of asserted facts. These facts are labeled according to whether they relate to the ideal or bugged code, and are numbered in the order of their assertion. They contain symbols specifying the interpreter action which gave rise to the fact along with the current goal and clause whose resolution was being attempted. This method is an adaptation of that used by Eisenstadt (1985) to facilitate a form of tracing and debugging known as 'retrospective zooming'.

In order to determine the effect of a particular bug in a given set of clauses both the ideal and the bugged code are run with the same initial query. The two outcomes and the two execution histories are then analysed to see firstly where (or if) they differ, and secondly which of the allowed bugs could account for any differences found. The analysis of differences is carried out by sets of operators designed to identify the specific conditions which result from the bugs of the relevant tree (see above). Each operator can explain the diagnosed bug to the student by means of a template which can be instantiated to produce descriptions of the effect of the current bug in terms of VIPER's viewpoints.

If the correct operator succeeds the system will have available, for a given initial query, the ideal and the bugged outcome, a statement of what constitutes the bug, and execution histories for both ideal and bugged code. These may be used as the basis for a range of tutorial activities.

The pedagogical goal of the system and the architecture described above suggest three essential tasks for the student. The first is to be able to describe or predict the bugged execution in terms of the models of execution. The second is to identify where the bugged execution diverges from what would be expected for a correct solution to the problem and thus to identify which clause contains the bug. The third is to identify the bug and describe its effects. These three tasks form the basis of three dialogues that VIPER uses to effect its tutoring.

For the first task the student is asked to describe each step of the bugged execution in terms of the appropriate models. (We use 'models' rather than 'viewpoints' here, as the description of execution is not carried out to achieve any particular purpose in the domain, such as the localisation of bugs. This means that the full 'viewpoint' apparatus which includes heuristics determining the applicability of different models to a problem is not required. The execution description exercise is carried out to check and rehearse a student's knowledge of the models before they are used to actually locate and describe bugs in the second and third dialogues). The student does this by making a series of menu selections each of which identifies a part of an execution model. The student's answers are checked against the system's execution history and, if necessary, can be corrected. The action symbol, goal, and clause contained in each line of the execution history allow the generation of correct answers and explanation.

Each of the interpreter action symbols is associated with a particular part of a model and with a corresponding menu choice. For a given execution step the correct answer (ie. menu option) is determined by looking up the menu choice that should correspond to the action symbol in that history step. Explanation is provided by expanding an asserted fact in the execution history into a more accessible account which is phrased in terms of the models of execution. Partial explanations, relating to a specific model or combination are generated by expanding only those trace facts which contain execution symbols relevant to the specified models.

The student's responses could be monitored for evidence of known misconceptions. This possibility introduces the need for some form of student model. A simple student model has been implemented for this execution-description task which records the number of correct, potentially correct, and wrong answers for each part of each execution model. This is not seen as contributing to the research interest of the system.

The second task for the student involves identifying where the bugged execution diverges from what would be expected for the ideal solution to the problem, and thus deciding which clause contains the bug. In the case of a variable being returned with the wrong value, this question breaks down into two others relating to separate events. If the bug causes the resolution which would give the correct value to fail, a second resolution has to succeed allowing the wrong value to be returned. The two questions are thus: a) where does the correct value fail to get instantiated? and b) where does the wrong value get instantiated? In each tutorial the system must make clear which question is being asked, as different descriptions and explanations are associated with each one. (Where the bug causes the 'wrong' resolution to succeed before the correct value could be instantiated, the two events described above are in fact the same single event and only one question may be posed).

The students are shown the bugged code and are asked to indicate which clause contains the bug. They are told that the correct answer is provided by an 'ideal' code solution (which they cannot see), and that only a single difference is allowed between this 'ideal' solution and the visible bugged code. (The need for these simplifications is discussed in chapter 6). This single difference must, of course, constitute the bug. In order to check an hypothesis about this bug the students may ask questions about the 'ideal' code via a 'Questions' menu. The same menu allows them to propose a candidate bug or to request a full explanation of the bug and its effect. This menu is also available in the third dialogue.

For the third task the student is asked to describe the effect of the bug and then to choose the relevant bug from a list of candidates. The student's description can then be checked against a keyword template assembled by the relevant bug-finding operator. Variables in this template are instantiated to the appropriate clause and goal, and the description is given in terms of the relevant models. These descriptions are given coherence by a set of explicit conventions which govern the mapping of the models onto VIPER's simplified bug catalogue. Where necessary, an explanation of the current bug and its effect can be

provided via the 'Questions' menu described above. This explanation is provided by operators which detail the inferences that can be made with each model in order to localise bugs.

The more detailed descriptions given in this section are intended to illuminate the points made in section 1.2. The importance of the viewpoint operators can be shown by briefly considering each of the dialogues described above.

In the first, 'execution description' dialogue, the student merely has to select the appropriate part of each model to apply in order to describe each step. VIPER performs this task by retrieving the symbol from the relevant execution history fact, and then retrieving the stored value (model part) associated with that symbol. This action illustrates the use of the first class of operators defined, those which simply retrieve a specific explicit piece of information from a model or execution history. Simple as it is, this operator allows VIPER to check the student's prediction for an execution step; to provide the correct answer if this is required, or else an explanation in terms of the bound values at the relevant point in the execution; to demonstrate the skill of describing execution, (when the operator is used repeatedly to describe a sequence of steps); and to describe the execution in terms of a single or of multiple viewpoints.

The second dialogue asks the student to identify the bugged clause and invites them to check their hypotheses about the difference between the ideal and bugged code by requesting information through the 'Questions' menu. This information is provided by other examples of the 'access' operator which simply retrieve stored values such as the functor of clause n , the number of clauses in the ideal code, or the name of the bug detected by VIPER.

Where the student requests an explanation the second form of operator is brought into play. This operator relates two explicit statements via an inference procedure and thus makes explicit information which is otherwise only implicit.

An abstract example would be: (A -> B, B -> C) -> (A -> C). There may be as many instances of this operator as there are inference procedures, and the inference procedure need not be logically correct. The operator seeks to define the bug by identifying a characteristic in the bugged code which is associated with the success or failure of a particular attempted resolution, and stating that this implies that the wrong result is obtained (first statement). The fact that the wrong result is obtained implies that the identified characteristic is different in the ideal code (second statement). Thus the presence of the bugged code characteristic coupled with the result of the specific resolution attempt implies that the characteristic is different in the 'ideal' code (conclusion). Operators of this kind are defined for each bug in such a way that only the correct one can succeed given the restrictions imposed on the domain. VIPER thus identifies the bug in terms of an inference procedure which is defined in response to observations of human reasoning, and which attempts, however inadequately, to have some relationship to that reasoning. The inferences made in the use of this operator are used to provide the explanation of the bug.

Both forms of operator discussed here are thus central to the design and execution of the system's dialogues. The simulation study described in section 1.1 identified a third form of operator which added information to, or deleted information from, the models, so that different inferences could be made, or different information retrieved. VIPER was not developed to the point where this operator could be implemented although certain uses for it were envisaged, such as transforming the Search Strategy model to represent a more advanced description of Prolog's behaviour, or perturbing a model to represent a student's misconception.

While the precise value of these operators is a matter for further debate, it is clear that the functionality that they provide could be generalised to other domains where viewpoints were implemented in similar ways, and where the strategy of generating an execution history was also adopted.

As has been stated above, the heuristics which state a viewpoint's area of application are not implemented explicitly in VIPER, but are implicit in the structuring of the domain. As a result of this, we cannot show VIPER explicitly adapting to individual students in the light of these heuristics, in the manner outlined above. The system was not, in fact, equipped with the complex student models which such adaptation would require, although a discussion of "further work" indicates how these models could be implemented using well-researched techniques. What can be shown, in the evaluation of the system, is that individual students would choose to work with a specific viewpoint in order to satisfy specific goals, such as improving their appreciation of the kinds of bug that can be trapped using the "Search Strategy" viewpoint. This is taken to imply that although the heuristics linking specific viewpoints with specific areas of application were not formalised explicitly for VIPER, the notion of linking viewpoints to goals and problems in this way is at least very useful in exploiting the adaptive potential provided by the use of multiple viewpoints on a domain. We see no reason why this kind of adaptivity should be limited to the domain explored by VIPER.

1.4 Design Considerations and Claims for VIPER.

The educational philosophy which has guided the design of VIPER's tutorial dialogues is that of Cognitive Apprenticeship (Brown et al. 1989, Brown 1989). This philosophy indicates that the purpose for which the knowledge being learned is to be used should be borne in mind at all stages of a tutoring system's design. It indicates that the exercises and practice which a system requires of a student should be as 'authentic', or as close to those of real practitioners, as possible. VIPER uses the tutorial tactics of Cognitive Apprenticeship: modeling of the target skill, the provision of a 'scaffolding' of threadbare concepts, an emphasis on different possible problem decompositions, and extended general practice.

On the basis of VIPER's implementation and evaluation we would wish to make several claims whose validity we believe we have demonstrated. The first of these is that the viewpoint formalism of models and operators allows the implementation of pre-defined viewpoints in such a way that the resultant systems can perform the relevant tasks in the domain. A second claim is that the strategy of using a meta-interpreter or simulation to produce a history of some process which is sufficiently detailed and structured to support tutoring in terms of multiple viewpoints on the relevant domain, is a robust and flexible method of implementing systems which can tutor in terms of the multiple pre-defined viewpoints. The use of this method requires that the representational requirements of the different viewpoints, and of the tutoring mechanisms which are to exploit them, should be considered at all stages of the design process.

Some weaker claims can also be made. These depend on further work being done to augment VIPER but we have demonstrated that these augmentations are quite feasible and only require the use of techniques which are established in the literature. Firstly the basic VIPER system could be augmented to carry out diagnosis in relation to inadequacies and misconceptions in the student's knowledge. This could be achieved in relation to each viewpoint by using unsophisticated student modeling techniques. Another weak claim is that VIPER's domain could be extended to include other pre-defined viewpoints on Prolog which could be implemented in the manner outlined above. This would not require any changes to the basic architecture of VIPER, and would use the same meta-interpreter and execution history.

A third weak claim is that, since any different viewpoints that were implemented would be suited to different goals and experience, an augmented VIPER could adapt its tutoring to these factors (assuming it had some representation of them) by choosing an appropriate viewpoint. For example if we wished to introduce the student to the models of Prolog that *are* implemented in VIPER, it would seem pedagogically wise to begin with the model which is closest to something that the student already knows: ie. the tutoring is related to

their previous experience. The system could simply ask for information about this. Where the student has some knowledge of theorem proving for example, we may begin with the Search Space model which, as defined in chapter 6, has many similarities to traditional theorem proving. The complexities of the Search Strategy and Resolution models could then be approached later. Even where the student was familiar with all of the models the bugs and exercises chosen for tutoring could initially focus on search space issues. Novices frequently wish to concentrate on particular aspects of the language, as our evaluation shows. They may, for example, wish to improve their understanding of resolution in Prolog. The goal may be satisfied by choosing problems and exercises which stress the use of the Resolution viewpoint. It is assumed that such adaptation increases the student's engagement with the system, and makes the tutoring more meaningful to them. This desire to adapt to the goals of students means that the system design process must take account of the possible uses of the knowledge to be tutored from the earliest stages.

This kind of adaptation to the misconceptions, goals or experience of the students can be characterised as tutoring 'with' the viewpoints; ie. particular tutorial strategies are realised through the use of different viewpoints. We can also describe a process of tutoring 'about' the viewpoints. The goal of such tutoring is that the student should appreciate that a given viewpoint is suited to the solution of particular classes of problems, and that it stands in a specific relationship to other viewpoints. If we again consider the electrical and mechanical views of a car ignition system, it is clear that they can both solve different classes of problems. They are however closely related, and there may be many situations which require their use in combination. Alternatively we may say that if the possible causes of a fault implied by one view are excluded, then one of the causes implied by the other view must apply. This implies that the student must learn to think both 'with' and 'about' viewpoints and must, at least initially, engage in some meta-cognitive activities to consider whether they are applying the appropriate view to a particular problem.

The chosen structure for viewpoints can be related to these considerations. The use of an explicit model which is interrogated by sets of operators allows the system to tutor 'with' a given viewpoint; ie. the system tutors, and can perform the tutored task, in terms of that viewpoint. This, in conjunction with the use of an execution history, allows the system to employ such tutorial tactics as demonstration, critiquing, and explaining. With further work, we claim that the system could also model student errors.

The inclusion of a set of heuristics in the viewpoint structure which detail its area of application allows the system to tutor 'about' the viewpoints in terms of their area of application and their relationship to other viewpoints. Such tutoring could be intended to remedy misconceptions in the student's knowledge of a viewpoint's applicability, or to promote some meta-cognitive appreciation of how the viewpoint is to be applied. These heuristics can be related to possible goals that a student may have, so that the system can adapt the viewpoint it uses for tutoring to those goals, where knowledge of those goals is available.

The adaptation of the tutoring to a student's previous experience is not dependent on any specific part of the structure described, but is a general form of adaptation implied by the use of multiple viewpoints. The system would require some specific knowledge relating its available viewpoints to other areas of study or other viewpoints. A specific viewpoint would be selected as being appropriate to the tutoring of a given student in the light of this knowledge.

Another weak claim of the thesis is that other, quite different, domains which require the use of multiple viewpoints can be implemented using the viewpoint formalism and system architecture described above. A detailed example of how this could be achieved is given in chapter 9 (the discussion chapter of the thesis) in relation to WHY (Stevens et al. 1979). Essentially the method involves building a simulation of the domain to be tutored and using this to produce an execution history for that domain. In this sense, VIPER's meta-

interpreter is seen as a *simulation* of a subset of Prolog. For the new domain the execution history must be sufficiently well-structured and sufficiently rich to support the range of viewpoints that the tutoring demands. Each of these would be formulated in terms of the model, operators and application heuristics described above.

1.5 Conclusions.

VIPER exemplifies a design methodology for formalising viewpoints and for implementing systems that can tutor using multiples of such viewpoints. This methodology is best suited to procedural domains, and requires that the potential uses of the knowledge, the intended methods of tutoring, possible student goals and possible previous experience be taken into account at all stages of the design process.

This chapter gives an overview of the thesis concentrating on the goals and motivations of the author. It considers the importance of viewpoints in tutoring systems, a particular method that has been chosen to pre-define them for implementation in a tutoring system, and the development of this method. The role of viewpoints in the choice of an example domain and in the design of VIPER are considered, along with more general reasons for the ultimate choice of domain. The interplay between the adopted structure for viewpoints, the structure of the chosen domain, and the design of the system is described. The domain formulations and the dialogues that they support are given in outline. The educational adaptations that the adopted structure for viewpoints makes possible are discussed, along with the need to tutor both 'with' and 'about' the viewpoints. The claims made in the thesis are summarised.

1.6 The structure of the thesis.

We conclude this chapter with an outline of the ten chapters which constitute the thesis.

Chapter 1. Introduction to the Thesis.

This chapter gives an overview of the thesis concentrating on the goals and motivations of the author. It considers the importance of viewpoints in tutoring systems, a particular method that has been chosen to pre-define them for implementation in a tutoring system, and the development of this method. The role of viewpoints in the choice of an example domain and in the design of VIPER are considered, along with more general reasons for the ultimate choice of domain. The interplay between the adopted structure for viewpoints, the structure of the chosen domain, and the design of the system is described. The domain formulations and the dialogues that they support are given in outline. The educational adaptations that the adopted structure for viewpoints makes possible are discussed, along with the need to tutor both 'with' and 'about' the viewpoints. The claims made in the thesis are summarised.

Chapter 2. Viewpoints in Tutoring Systems: Uses, Structures, and Domains.

This chapter consists of a literature review which introduces the relevant areas and gives a first formulation of the problem, the research goals, and the intended research direction.

Chapter 3. A Formulation for Viewpoints.

Chapter 3 considers the various factors involved, and produces an outline specification of a formalism for implementing viewpoints. This is related to the educational philosophy of Cognitive Apprenticeship.

Chapter 4. Testing the Formulation: a Protocol Analysis.

This chapter describes a protocol analysis study that was undertaken to test and refine the viewpoint formalism described in chapter 3. Protocols are obtained which display quite distinct patterns of inference when different groups are asked to apply different mental models to the same simulated system.

Chapter 5. Testing the Formulation: Formalising the results of the protocol analysis.

The single formalism for viewpoints which is described in chapter 2 is used to formalise both of the patterns of inference identified in chapter 4. Parts of the formalism are developed in greater detail.

Chapter 6. The Implementation Domain and Tutoring Goals.

Chapter 6 contains a detailed formulation of the implementation domain. First the models of execution are developed, and then mapped onto a restricted catalogue of bugs. This mapping is achieved by specifying a number of conventions which produce a simplified and consistent debugging 'world' which is suited to the needs of novices.

Chapter 7. VIPER: Viewpoint-based Instruction for Prolog Error Recognition.

This describes the implementation of VIPER itself. The main structures described are the meta-interpreter, the code which produces and analyses an execution history, and the predicates which manage the three forms of dialogue that VIPER can support. These dialogues are structured in terms of the main tasks in the domain.

Chapter 8. Evaluating VIPER.

This chapter describes an evaluation of VIPER by seven students with encouraging results. The system was configured to give a standard tutorial where the student first practised the application of the models to describe execution, and then practiced

debugging in the simplified domain by choosing which viewpoint they wished to work on.

Chapter 9. A Discussion of VIPER.

This is the main discussion chapter. The research goals are reviewed and an assessment made of what has been achieved. The usefulness of Cognitive Apprenticeship as an educational philosophy which supports the design of systems using multiple viewpoints is also assessed. A range of future work and possible system developments is described.

Chapter 10. Conclusions.

Where chapter 9 provides a detailed discussion and conclusions, this chapter summarises the main claims that can be made on the basis of the work described in the body of the thesis, and the justifications of those claims.

Chapter 2. Viewpoints in tutoring systems: uses, structures, and domains.

> "The multiple-viewpoint framework allows us to conceptualise the whole question of knowledge-driven CAI in a new way, in terms of how knowledge should be partitioned among different viewpoints" (Stevens, Collins and Goldin 1979. Quoted from the 1982 reprint p. 23).

The literature of Intelligent Tutoring Systems (ITS) contains many references to the virtues of using multiple representations of the domain knowledge in order to express different viewpoints upon that domain. This literature review examines relevant past work in the field, and then notes some of the educational issues involved. Subsequent sections of the chapter deal with mental models, Prolog tutoring and debugging for novices, and explanation in relation to the use of multiple viewpoints. These are variously relevant to studies and implementations described in later chapters. The final section of the review describes a philosophy of education which supports the use of multiple viewpoints in tutoring.

The problems on which the chapter focuses may be stated as follows: if ITSs need viewpoints, how are we to conceptualise them, how might they be implemented in our systems, what goals would this serve, and what implications might it have for our overall design philosophy?

2.1 Viewpoints in ITS design: the problem.

The use of multiple viewpoints has been considered desirable for a number of reasons which can be illustrated by reference to previous work in ITS. Four core issues are discussed in parts 2.1.1 - 2.1.4 of this section, (2.1). These sub-sections refer to previously-implemented systems to illustrate the influence that the issues relating to multiple viewpoints have had on system design.

The core issues, listed by the number of the section dealing with them, are:

2.1.1 The need to use multiple viewpoints in order to correctly carry out a single activity.

2.1.2 The need to use different viewpoints in order to carry out different activities.

2.1.3 The need to employ different viewpoints at different points along a learning path.

2.1.4 The possibly contrasting needs for system efficiency in task execution, and clarity in explanations given to the student.

A fifth issue, the desire to promote meta-cognition and reflection in the student, is dealt with in parts 2.1.5 and 2.1.6 of section 2.1. These parts are also concerned with more general issues affecting system design.

2.1.1 Different viewpoints required for the same activity.

The earliest and clearest explicit statement of the need for multiple viewpoints in tutoring systems is found in the work of Stevens, Collins and Goldin (1979), when they discuss the limitations of the WHY system. In order to tutor the causes of heavy coastal rainfall, they implemented the domain representations of the system in the form of 'scriptal' knowledge, which defined the sequence of conditions leading to a downpour and described such entities as a warm airflow and warm water mass. This was found to be inadequate however, as students' accounts of the process after tutoring contained a number of bugs representing the importation and incorrect use of concepts external to the intended domain. An example is the 'sponge' bug, which explains heavy rainfall in terms of the moist air mass being 'squeezed' against the coastal mountains. They concluded that it would also be necessary to represent and tutor 'functional' knowledge of particular processes, such as condensation, and of such generalised relationships as 'inverse processes' and 'feedback systems'. They also make the point that these differing viewpoints would have to be

properly integrated and updated if other bugs were to be avoided. The argument here is that if students are to acquire knowledge about this domain without bugs or misconceptions, then different viewpoints on the domain will have to be tutored and integrated.

Stevens and Collins (1980) discuss the issue of viewpoints, (here 'views', or 'points of view') in a more general context. They discuss viewpoints in terms of alternative 'models' which can be applied to a given situation to form hypotheses and make predictions. They conclude that specific strategies must be available to determine when to use a given model, and also to determine how this model relates to the others that might be used. This is seen as having major implications for education generally, and as necessitating a move away from 'static' or 'surface' forms of knowledge representation such as that used for WHY (Stevens, Collins and Goldin 1979).

A different combination of viewpoints is shown to be necessary by the work on the successive versions of the SOPHIE system. This system was designed to train students in the repair of electronic circuits by having them find the faults in a simulated circuit. In SOPHIE I, (Brown and Burton 1975) the student may advance a hypothesis and ask for feedback. This hypothesis is evaluated by comparing the current values it predicts with those found in the simulations of a working and a faulted circuit. (A numerical model of the circuit provides quantitative data about the various states within it). This data is interpreted by "inference specialists" which embody the laws of electronics. This allows hypothesis generation and testing, but not causal explanation. The designers hoped to achieve such explanation by means of another module which contained qualitative knowledge, such as which components are most likely to fail, how power amplifiers may be stressed, and heuristics about how to combine this knowledge with the quantitative data.

Their ambitions were partly realised in SOPHIE II (Brown et al. 1976). Here, a troubleshooting expert demonstrated how it solved a problem by qualitative reasoning. It

attempted to make its measurements causally meaningful, and to explain its strategy en route to a solution. In summary, both qualitative and quantitative analyses of the same electrical circuit were needed in order to provide a tutoring episode which contained both hypothesis testing and causal explanation. This may be described as a need for complementary modes of analysis, or, more loosely, as a need for different viewpoints.

For SOPHIE III, (Brown et al. 1982), it was intended that yet more viewpoints should be added, with the intention of providing active coaching in relation to the student's performance. The design called for an 'electronics expert' to draw on both the quantitative and qualitative knowledge of circuit behaviour so as to build up a database for a 'troubleshooting expert'. This troubleshooter was to have circuit-independent knowledge of how to manage a set of fault hypotheses, and how to propose new measurements to the electronics expert.

Yet other kinds of viewpoints may be found in NEOMYCIN, (Clancey 1983). The ancestor of this system, (MYCIN, Shortliffe 1976) was not a tutoring system, but an expert system designed to diagnose bacterial infections. It was, however used as the basis (domain representation), of GUIDON, (Clancey 1979), and it was this exercise which led directly to the development of NEOMYCIN. MYCIN attempted to formalise the knowledge of medical experts in a series of condition-action, (if ---- then ----) rules known as productions. The order of these rules, and the order of their subgoals, was critical to successful diagnosis. Each one that fired contributed to the database of evolving hypotheses. There was no explicit problem-solving strategy, (beyond exhaustive search), built into the system. Rather, it ploughed exhaustively through the same set of rules for each diagnosis. Although it was quite successful at problem-solving, its usefulness was limited by an inability to give convincing explanations of its conclusions.

Clancey (1983) describes how he and his colleagues at first saw themselves as "applications engineers" whose task was to adapt MYCIN's explanation facility to a tutorial

setting. It was, however "--- surprising to find out how little the explanation facility could accomplish for a student." (ibid. p. 217). MYCIN did not solve problems in the same way as a human expert. Its production system rules were formulated to diagnose infections, which it did successfully by exhaustively searching the problem space. There was thus no explicit problem-solving strategy which could be taught to a student. As Clancey freely admits, "Focusing on a hypothesis and choosing a question to confirm a hypothesis are not necessarily arbitrary in human reasoning", (ibid. p. 220), thus raising serious questions about the usefulness of MYCIN as the basis of a system for tutoring students in diagnostic reasoning.

The relevance of this to the current discussion becomes clearer when one examines the ways in which Clancey tried to overcome this problem. The re-evaluation occasioned by GUIDON's failure led to the development of NEOMYCIN (Clancey 1983). After listing the kinds of knowledge used by human experts for diagnostic reasoning, a series of "meta-rules" were developed to give meaningful access to the domain rules, (ie. to encode the notion of a hypothesis), and to manage and interpret a changing list of hypotheses. Similar structures were developed for GUIDON, known by such names as "rule models" and "rule schemas". Their purpose was to make clear the relationships between the rules of a domain, so that a reasoning strategy could be learned. It can be argued that they still leave out a great deal of crucial procedural information, such as why the subgoals of each rule are considered in a particular order. The main point, however, is that the various meta-rules and their sub-structures may be seen as alternative viewpoints on the domain, without which any useful tutoring or explanation is not possible. This very point is made by Wenger (1987 pp. 275-7) who describes NEOMYCIN's substructures as "orthogonal viewpoints", which "--- act like windows, giving access to the knowledge base". He concludes that these

> "--- access channels are crucial to support the variety of reasoning tasks and focusing strategies by which data and conclusions are connected". (Wenger 1987 p. 276).

2.1.2 Different Viewpoints for different activities.

The previous sub-section described situations where a combination of different viewpoints was necessary for a given activity to be successful. The argument in this sub-section is that specific activities can strongly condition the way we need to perceive an object or system, and that this may imply the adoption of a specific viewpoint to support that activity. An everyday example could be the metallic body of a car being seen not in terms of its shape and mechanical functionality, but in electrical terms as an 'earth' connection. In short, the way we see things may depend on what we want to do with them.

A similar example exists in the ITS literature. Knowledge elicitation carried out in preparation for the construction of STEAMER, (Hollan et al. 1984), showed that the way experts think about a particular machine does not necessarily depend on its physical components, but on the actions they wish to perform with it. Accordingly the designers attempted to base the tutoring process on the conceptual abstractions, (mental model), of the experts, rather than on the precise physical characteristics of the machines, and labelled their approach "conceptual fidelity". The system was built to tutor the operation and maintenance of steam propulsion plants, and was designed as a graphical interface to an interactive, inspectable simulation of such a plant. This core simulation had already been built. An example of the use of a function-specific viewpoint occurs when the complex assembly of parts which makes up a working turbine is labeled by experts simply as a 'steam chamber with drains'. The safe and efficient running of this "abstract" machine requires some "abstract" procedures, such as opening the drains before admitting steam to the chamber, (Stevens and Roberts 1983). This example may be used to draw attention to the function-dependent or "context oriented" aspect of viewpoints: their use may be dependent on a specific context or set of goals.

STEAMER makes use of this by offering over one hundred different views of the plant in question, from abstractions such as 'the basic steam cycle', to control panels describing

some specific state of an operating system. A pump may be seen as an instance of a 'positive displacement pump' or as a component of a 'pumping station'. The richness of the information provided by the simulation is exploited by a frame-based representation system which accesses the different combinations of data required to support different viewpoints. A specific object's relationship to the system may be described and explained in terms of its physical connections, energy connections, information connections, or perhaps in terms of the procedures which operate upon it. Trials with trainees have convinced the designers that the systems ability to change viewpoints is crucial to its effectiveness.

2.1.3 Different viewpoints from novice to expert.

STEAMER (Hollan et al. 1984) showed a preoccupation with 'conceptual fidelity' in the interface used to access a pre-existent simulation. With QUEST however, (White and Frederiksen 1986) this concept was applied to the system's most basic domain representations. QUEST was designed to tutor the behaviour of electronic circuits. Like STEAMER it was conceived as an inspectable simulation with a graphical interface, but the underlying representations form a sequence of qualitative models based on a causal calculus. Rather than concentrating on a single device, QUEST's successive models corresponded to increasing levels of expertise in the principles of the domain.

The models differ along the dimensions of type, order, and degree. 'Type' describes such characteristics as 'qualitative', 'proportional', (eg. descriptions such as "less than" or "equal to"), and quantitative, while 'order' deals with the derivatives that the model uses to define changes in a circuit. If one order describes any change of voltage, for example, the next may describe the rate of that change. The 'degree' dimension adds to the complexity of the model by taking additional constraints into consideration. The models are thus generic, each relating to a given level of complexity in the analysis of electronic circuits. A crucial connection between the models was 'upward compatibility', which ensures that a

simpler model can be extended or refined so as to match the next more complex one with a minimum of effort. This system was quite successful in tutoring a small group of students in a restricted problem-solving area.

White and Frederiksen do not claim that the model structure they define provides on its own an adequate model of expertise, since they believe that this consists of much more than the ability to predict specific states. They regard it as crucially important that different models of various type and order be integrated into a coherent and effective understanding of the domain. This 'integration', and the question of precisely how particular models are related to the solution of particular problems, remain as topics for future research for them. These conclusions echo those of Stevens, Collins and Goldin (1979). If we regard the different model types as different viewpoints, we may conclude that such viewpoints may be implemented in a tutoring system, and that they may form a pedagogically useful progression from novice to some level of expertise.

2.1.4 Glass boxes and black boxes

The origin of the 'glass box' and 'black box' terminology may not be entirely clear, but an early manifestation can certainly be found in du Boulay et al. (1981). The point of the terminology is that different models of a system may be presented according to the needs or expertise of the user even when tutoring is not the immediate goal. The inner workings of, in this case, a computer language interpreter are not available for inspection and are regarded as a 'black box'. This may present little problem to an expert, but novices attempting to program in the language may soon find that their interactions with the system become unintelligible as they have no idea about what state the system is in. The problem may be alleviated by providing a 'notional machine' as an idealised model of the system which can describe the more important events occurring inside the 'black box' in terms familiar to the novice. Statements about "stack overflow" for example, might be meaningless, while a statement concerning "store size" might communicate usefully. This

'notional machine' is the glass box. Different models are thus provided for expert and novice.

While du Boulay et al. (op. cit.) are concerned primarily with programming languages, we may extend the idea to tutoring systems. The comments above in relation to GUIDON and SOPHIE (section 2.1.1) and QUEST (section 2.1.3) describe systems whose knowledge representations attempt to capture some degree of "psychological validity", in the sense that they attempt to solve problems or make inferences in ways which are meaningful to humans. The meta-rules of GUIDON (Clancey 1983) are designed to encode the notion and form of hypotheses. The modules of SOPHIE (Brown et al. 1976) and the models of QUEST (White and Frederiksen 1986) are intended to exhibit qualitative reasoning. The point of this is that for tutoring purposes, the systems should be able to explain their inferencing in terms that humans might use or understand.

WEST (Burton and Brown 1979) shows that this and other pedagogical aims can be achieved without having the machine reason in those terms. Where the system presents the student with a task or problem, the system may solve that problem for itself by means of a 'black box', but, having found the solution, carry out its tutoring and explanation in terms of a 'glass box'. WEST is based on a game which exercises arithmetic skills. A player is given three numbers at random and has to compose an expression which is evaluated to give the number of moves towards the goal, their 'home town'. Players may also aim to 'bump off' other players or to take short cuts.

Tutoring is based on 'differential modeling'. The student's performance is compared to that of the system's expert, and diagnosis starts if the student's move is not optimal. This design strategy is necessary due to the random nature of the game, ie. is not possible to say in advance what skills will be required at any specific point in the game. The various possible moves are analysed in terms of 'issues' or strategies (eg. trying for a 'bump') to

identify those that the student is using and those that reveal weaknesses in their play. The system does not represent interactions between issues or erroneous issues.

Because the domain of the game is limited, the expert can work by exhaustively listing all possible expressions and simulating their application to the current state of play. This expert thus needs no representation of the issues. Its moves are analysed by the same diagnostic procedures that are applied to the student's moves. However, once the ideal move has been identified, it may be presented to the student in terms of the related issue. Two quite distinct representations (viewpoints) are thus exploited in conjunction so as to effectively tutor a single domain. As Wenger (1987) points out, the system expert has considerable performing power, but is not "psychologically plausible". The 'issue' recognisers have no performing power, but are able to justify the expert's moves in "teachable" terms. Wenger also points out that in more complex domains, where the computational costs of analysing the expert's move may become prohibitive, the domain expertise would have once again to be implemented in terms of the pedagogical issues that the system was designed to address. Nevertheless, WEST shows that a 'black box' domain expert can be pedagogically useful if a suitable 'glass box' representation can be found to complement it. This point is of direct relevance to the tutoring system which is being proposed in this chapter.

2.1.5 Exploring the student's viewpoint.

In the consideration of viewpoints and tutoring, it may be argued that we should encourage students to develop and explore their own views of a domain, and that tutoring should be adapted, where possible, to the student's view. The Alternate Reality Kit, ('ARK', Smith 1986), while not strictly a tutoring system, does move in this direction. ARK is an environment designed to express physical laws, (correct or otherwise). It contains a wide variety of objects, all of whose behaviours may be programmed or connected, and all of which obey the laws set for the environment as a whole. The intention is to allow students

to appreciate why the laws of physics are as they are by seeing what happens when they are broken. Students may set the system up to reflect their own view of the behaviour of a physical process, and observe the consequences; (O'Shea and Smith 1987). While this system does allow students to express their own viewpoint on a domain, it does not carry out any active tutoring or student modeling, and is perhaps best characterised as a system which promotes exploration.

2.1.6 Viewpoints and design philosophies.

The systems discussed in the previous sections indicate that the question of viewpoints is an important one for ITS design. If viewpoints are to be incorporated into the systems, we may ask what general implications this may have for our design philosophy.

A standard reference for the state of the art and for the most general approach to tutoring system design is Wenger (1987). The central idea of this work is that of 'knowledge communication', which is defined as:

> "... the ability to cause, and/or support the acquisition of one's knowledge by someone else, via a restricted set of communication operations." (Wenger 1987, p. 7).

It may be argued that this approach has a number of conceptual limitations since it implies that some single, pre-existent mass of knowledge has to be carefully poured into a politely passive student. Taken to its extreme, the student would be seen simply as an extension of the computer program. There are those (see section 2.6) who believe that the student is more properly conceptualised as an active participant who is engaged in the tutoring process, and whose learning involves the structuring and integration of the information with which they are presented rather than its passive acceptance. This in turn implies that not all students may structure it in the same way. These points seem to be acknowledged later in Wenger's book. He stresses, with reference to Lave, (1988), that the student must be actively engaged in problem solving, in order that they perceive specific problems,

discover the limitations of their current view, and "... *demonstrate a new viewpoint's conceptual superiority* ". (Wenger's italics. Wenger op. cit. p. 411).

Several points can be made here. Firstly, while Wenger is happy to emphasise the importance of the student actively acquiring new viewpoints, this activity is not well described by the concept of "knowledge communication" as defined above. Also, any system which was attempting to move the student from one viewpoint to another would have to have different accounts of the domain available to it, ie. would have to have multiple viewpoints on that domain. This would require more than a "communication" model as some decision processes for choosing, tutoring, and using the appropriate viewpoints would also have to be defined.

There are other indications that Wenger really intends his readers to see the student as a more active participant in learning. He proposes (p. 321) the notion of "equivalence classes" of models of knowledge, "... within which communication is possible and useful." This sounds far more complicated than, (and quite distinct from), "... the acquisition of one's knowledge by someone else... ", but he gives no further information about how such equivalence classes might be derived, what they might look like, or how, precisely they might be used. Elsewhere he states that:

> "... recipients must interpret communication by a process of reconstruction, and there is always some uncertainty about the similarity of the knowledge possessed by each participant." (Wenger op. cit. p. 321),

but the implications of this for 'knowledge communication' are not made clear. On page 365 Wenger states that it is obvious that a "model of communicable knowledge" need not be static, but could be modified by the communication process. Unfortunately he does not give any description of the mechanism by which this could take place. In the Epilogue Wenger states that:

> "... both the knowledge states involved in knowledge communication are modified: knowledge communication is viewed as a dynamic interaction between intelligent agents by which knowledge states are engaged in a process of expansion and articulation". (Wenger op. cit. p. 431).

This seems to bear little relation to the definition given earlier, which described the "acquisition" by some other individual of one's pre-existent knowledge. While this latter statement does seem to embody a more constructive view of the student, it comes late in the book and still leaves the reader with considerable doubt as to what 'knowledge communication' really entails, thus limiting its usefulness as a basis for design. The issue is not clarified by Wenger's own attempts to define viewpoints. These are discussed in section 3.1.

Similar dissatisfaction with the notion of 'knowledge communication' is shown by Self (1988a, 1988b) who suggests that the reliance on a single representation of the domain has led to an inappropriate "trinity" model of ITS design, (the student model, the domain representation, and the tutoring component). He suggests that this may lead to a rather authoritarian style of tutoring, which assumes that the system embodies in a Platonic sense, the one true representation of the domain. This being so, the approach may also lead one to assume that the student may be adequately represented as a 'subset' of the expert. Self (1989) indicates that 'education as transmission' is not an idea favoured by this century's educational philosophers and quotes Perkinson (1984) in saying that "... the transmission theory of education is both false and immoral".

Self suggests that we should more properly represent the domains as "belief systems", which may be reconceptualised, and that more attention should be paid to meta-cognitive skills, such as reflecting on one's own problem-solving processes. This does seem like a far more fruitful direction if our purpose is to explore viewpoints in ITS. Our relief should be restrained, however, as the area of belief systems is fraught with serious research problems.

Self (1990) has produced a wide-ranging review of systems which attempt to model the holding and revision of sets of beliefs, in an effort to list ways in which viewpoints might be structured and implemented. The problems thrown up by this review are legion. Agents may, for instance, hold inconsistent or contradictory viewpoints, requiring such formulations as 'local reasoning' (Fagin and Halpern 1987). The demands of computational efficiency and psychological plausibility may require that non-deductive reasoning methods be used. Other problems relate to the processing mechanisms used with viewpoints. If modal logics are used, then the belief operator 'B' becomes "referentially opaque", ie. even if we have proved that two expressions are equivalent, we cannot in the main substitute one for the other.

Another area of work concerns the revision of viewpoints, more generally known as 'reason maintenance' or 'belief revision' (Martins and Shapiro 1988). Where the viewpoints are formed incrementally from the accretion of a series of individual beliefs the problem is "relatively tractable" as the latest additions alone may be revised. Where there are radical differences between viewpoints however, the problems are more profound. Writers such as Kornfeld and Hewitt (1981) stress the need to preserve the "core of fundamental concepts" in "adjusting" such a viewpoint. The definition of the "core" and the means of adjustment are not made clear.

Self's review indicates that the construction and processing of belief systems is an area beset by fundamental technical difficulties. This being so, the issues surrounding their use in ITS must be equally unclear. With some thought however, it is possible to separate the issues surrounding the construction and use of viewpoints from those entailed in research upon belief systems. Rather than making a tutoring system based on belief revision the focus of our research, we may re-define the problem and concentrate on a system which utilises a number of *pre-defined* viewpoints. This should allow a robust system to be built, thus allowing an investigation of the issues which arise when we attempt to progress

beyond a 'knowledge communication' model of design. The research described in this thesis takes exactly this approach.

A rather different notion of 'viewpoints', giving rise to correspondingly different conclusions about tutoring system design, is voiced by Lesgold (1988). Lesgold is primarily concerned with the issue of curriculum organisation for tutoring systems. He contrasts "conventional instruction" and "intelligent instructional systems" in the following terms: "conventional instruction", while having an explicit curriculum, usually does not have a complete representation of the target knowledge. For "intelligent instructional systems" the emphasis is reversed. They generally have a "complete" representation of the target knowledge, but little explicit representation of curriculum or curricular goals. The 'viewpoints' that Lesgold describes are essentially different ways of accessing and ordering a set of defined lessons so as to serve different goals.

The notion that ITS systems have a "complete" representation of the target knowledge can be used to point out some of the differences between the notion of viewpoints developed in this thesis, and that proposed by Lesgold. The emphasis of sections 2.1.1 - 2.1.5 above is on the need to extend the previously-inadequate representations of the systems described, and to develop descriptions of the relevant domain in terms of viewpoints which are explicitly taught to the student. Even with such extensions, researchers such as White and Frederiksen (1986) explicitly state that proper 'expertise' involves more than their representations actually embody. Lesgold is less concerned with reformulating the domain that with organising a pre-existent set of lessons in terms of the pre-requisite knowledge required for the different kinds of problems that may be presented to the student. Thus for example, one problem-based curricular organisation, or 'viewpoint', may first emphasise physical laws, while another emphasises measurable properties or qualitative problems. It seems that for Lesgold, it is not these 'modes of analysis' which are themselves the 'viewpoints', but the collection of different parts of each of them which make up a path through the curriculum.

This said, it must be admitted that Lesgold shares some of the concerns of this thesis. He is concerned with the *application* of knowledge, and repeatedly states that real learning involves more than the sum of the lessons learned. For him this implies that remedial lessons must check the students ability to transfer knowledge to new problem-solving contexts, (something others may regard as a more fundamental issue), and that the limits of applicability for each piece of knowledge should be made clear. Lesgold also wishes to adapt instruction to the learning preferences of the students, and to build on their previous experience. He currently has little to say on the latter issue. His approach leads to a three-layered system architecture, where the bottom layer, the various lessons, are accessed in terms of a lattice of connected curricular goals. In the top layer, labelled as 'meta-issues', separate nodes form the starting-point for each 'viewpoint'. Little information is given about these, but they appear to involve such issues as catering to a given student's strengths and weaknesses in learning.

2.1.7 Conclusions: defining the problem.

What conclusions can be drawn from this review of tutoring systems which utilise different viewpoints? It is clear that in this context "Viewpoints" are not simply alternative representations. We may illustrate this by representing the same information, (eg. the novice's view of an electrical circuit), in two different formalisms. It may be represented as a semantic net, or as clauses in predicate calculus. In other words, a single viewpoint may be expressed as two different knowledge representations. Conversely, a single representation, such as the quantitative circuit model of SOPHIE 1, (Brown and Burton 1975), may be used by several viewpoints. In this case, a single circuit model is used by a number of "procedural specialists", to answer 'what if' questions, to evaluate the student's hypotheses, to list all possible hypotheses, and to evaluate requests for new measurements made by the student.

How are we to characterise what we might mean by a "viewpoint"? Minsky (1981) provides direction with a memorable example. He points out that for hard problems, one "problem space", (ie. an initial state, a goal state, and a set of operators), is not usually enough. If a car has a flat battery, we may suspect a fault with the generator. Describing this as a mechanical system, we see a pulley wheel driven by a belt from the engine. The armature is a rotating device associated with carbon brushes, held in position by bolts and screws. If we are trying to fix the system, this viewpoint implies a specific set of questions about belt tension, bolt security etc. There is, however, an alternative and complementary viewpoint, which sees the generator as an electrical system. The armature is now a flux-linking coil, while the commutator and brushes are a switching system. The entire metallic body of the car may be seen as a battery (earth) connection. This viewpoint also has its own set of diagnostic questions. While electrical faults generally require mechanical manipulation in order to rectify them, the isolation of the fault may well require that both modes of analysis be used separately. This implies that there is some other form of knowledge involved, a 'control knowledge' which allows decisions to be made about when and how each mode of analysis is to be used. This partitioning of the problem space allows the search for a solution to be carried out more efficiently, ie. in smaller search spaces.

The viewpoints described in the previous paragraph may be characterised in two ways. Firstly, they are complementary modes of analysis, both of which may be required in a particular context. Secondly, they refer to the same set of objects, but identify and structure them in quite different ways. The value of this seems to be that each view brings out particular features of the domain in question, and in doing so can reduce the search space for certain problems.

Our review of the literature can be related to this example. The remarks in section 2.1.1 indicate that multiple, complementary viewpoints may indeed be required to carry out certain tutoring and problem-solving functions. Section 2.1.2 shows that for given tasks a

specific viewpoint is most effective. In terms of Minsky's example, an electrical perspective would not necessarily be of use in the solution of a mechanical problem. Section 2.1.3 serves to remind us that there may be profound differences between the viewpoints of novice and expert, which probably comes as no surprise to your mechanic.

Given the preceding remarks, and the goal of building tutoring systems, we may frame the problem as follows: ITSs need viewpoints, so how are we to conceptualise them, how might they be implemented in our systems, what goals would this serve, and what implications would it have for our overall design philosophy? The question of whether or not our systems could allow students to work with their own idiosyncratic viewpoints is seen as premature. We may note, as with the generator example above, that some problems require the use of more than a single viewpoint to arrive at a solution, while certain tasks are facilitated by the adoption of specific viewpoints. This last point implies that for a viewpoint to be of use to us, we must go beyond learning the viewpoint itself to learn also how it is to be used. What is needed then, is an approach which emphasises and makes explicit this context-related aspect of our domain models. It is in this sense, as well as in building specific models, that the student has to "structure" or "reconstruct" their domain knowledge. This structuring involves learning when and how specific knowledge is to be used. Where a problem demands that one viewpoint must be used in combination with another, it also involves the explication of meta-knowledge about that viewpoint's relationship to other knowledge, as Self (1989) advocates. In other words such structuring involves the integration of viewpoints.

On the question of implementation, little may be concluded at this point, since each system studied appears to offer a different solution. An important point seems to be that while highly complex or very large domains require an implementation with some 'psychological plausibility', smaller or less complex domains can also be well served by a 'black box' allied to an appropriate 'glass box'.

If our design philosophy directs us to view the student as an active participant in the learning process, then this view can only be meaningful if the system we design adjusts to that student's participation. If our approach makes a range of viewpoints available for tutoring the same topic, then our system may seek to adjust to such factors as the student's knowledge state, learning history, goals, and preferences. 'Knowledge state' here can refer to the degree of integration of viewpoints as well as their levels of expertise. Where a number of viewpoints on a domain are implemented and available, decisions will need to be made about which one is to be tutored at a given point in the tutorial process, how it is to be tutored, and why. If we desire adjustment rather than imposition, it would seem preferable that these decisions be arrived at through a process of negotiation between system and student.

It is worth pointing out here that the necessary decisions relate to two distinct levels of analysis. Initially, a domain could be tutored in terms of a given (most suitable) viewpoint. Where the student is made aware that alternative viewpoints are available, and that tutoring is to be conducted in terms of these, then the student, and the decision-making process, must also consider the relationships between the viewpoints, and the differing areas of application for each one. This thesis is concerned with both levels of analysis.

2.2 Mental Models

How should we conceptualise 'viewpoints'? The review of ITS systems and design philosophies given in section 2.1 showed little agreement about what constitutes a 'viewpoint'. If we look outside the area of ITS for useful concepts a strong candidate would appear to be the psychological concept of the 'mental model'. Although this term is widely used, there seems to be a variety of conflicting definitions for it; (eg. Johnson-Laird [1983], versus Gentner and Stevens [1983]). Some of the clearest definitions appear in Young (1983) and are given below.

The intention of this section is not to review the whole literature of mental models, which is extensive, but to discuss areas of work which are directly relevant to our attempts to formulate and implement viewpoints in a tutoring system. The work of Gentner and Gentner (1983) and Kieras and Bovair (1984) indicates that inferencing can vary with the model used. Larkin et al. (1980) and Clancey (1985) pointed out the importance of some form of indexing mechanism to link the problem at hand to the knowledge which may solve it. We show this to be relevant to the position of writers such as Self (1989) and Brown, Collins and Duguid (1989) in their concern with meta-cognition and learning.

2.2.1 Studies of mental models.

An important paper is that by Kieras and Bovair (1984) who investigated the differences attributable to the presence or absence of a mental model when making inferences about the operating procedures of a previously unknown device. They studied the time and number of trials required to eliminate redundant moves from the procedures in the two conditions. Their evidence shows that the 'with model' condition is superior when a suitable device model is used. They deal with the actual inferences made by the subjects only in passing, and appear to assume that they will be the same as those developed in Kieras' (1984) simulation model. They conclude that a "suitable" device model for inferring operating procedures is based on concepts of device topology and power flow for the device in question.

Also relevant in this context is the work of Gentner and Gentner, (1983). In the paper entitled "Flowing Waters or Teeming Crowds: Mental Models of Electricity", these authors showed that different inferences result when different models of the domain are used. Working with a simple electrical circuit, they asked one group to consider it in terms of water flowing through the circuit, the other to consider it in terms of objects, (crowds of people), racing through passageways. When asked to answer questions about the effect of parallel resistors on current flow, the subjects using the "moving crowd" model showed a

better, (more accurate), performance than those using the water analogy. This is attributed to the suitability of the analogy for the problem posed. In terms of a moving crowd, parallel resistors represent two gates for progress rather than one, resulting in a greater flow rate. In the hydraulic model, resistors apparently represent drag, and more of them means more drag, (and thus less current flow), regardless of configuration.

The Kieras and Bovair study shows that performance in deducing the correct operating procedures is significantly affected by the presence or absence of a suitable mental model. The Gentner and Gentner study shows that different inferences may result from using different models. In dealing with results, however, this particular study does not attempt to systematically capture the inferential processes involved, and thus does not give detailed support to the assumptions about why the models produce different results. (This question is approached through these author's subsequent theories of analogy).

Let us assume that we wished to model the subjects' performance in these studies computationally, in terms of the notion of viewpoints outlined above. (ie. we require a performance model while psychologists debate the psychological reality of their various accounts). We would need to augment the Gentner and Gentner study, and the Kieras and Bovair study, to give a) some account of the means by which inferences are drawn from the models, and b) some account of how a particular model is chosen as being applicable to a particular problem.

We have chosen to focus on the first of these topics. The methodologies of these two experiments are combined for a study, (detailed in chapter 4) which observes subjects applying two distinct models to the operation of the same system. (The idea of operating a device is taken from the Kieras and Bovair study, while the comparison of two mental models is taken from the Gentner and Gentner study). The verbal protocols from these sessions are analysed in an attempt to distinguish different patterns of reasoning between

the two groups. These differences are modeled in terms of a structure for viewpoints (outlined in chapter 3), as a demonstration of the usefulness of this structure.

The two kinds of mental model which are used in the study are two of the most commonly distinguished ones, that is Structural and Functional models of a specific system. Young (1983) refers to these two types as "Surrogate" and "Task/Action Mapping" models, and explores the qualities of each. Where 'D' is the device in question, the Surrogate (Structural) model is:

> "... a physical or notational analogue of the mechanism of D, and can be used to answer questions about D's behaviour." (Young 1983 p. 38).

Young's "Task/Action Mapping" (or "Functional" model) is described as:

> "...the core of the mapping between the user's actions on D, and what D does." (Young 1983 p. 38).

(We shall refer to these two types in terms of the more commonly-used descriptions [eg. di Sessa 1986] "structural" and "functional"). Logically, the differing content of the two models gives them different utility. The ability provided by the "surrogate" model to reason about D's behaviour should allow one to make predictions about that behaviour. It may not, however, be immediately obvious how to achieve a specific goal using that same model. Di Sessa (1986) discusses what appears to be the same distinction in terms of "functional" and "structural" models and gives this example: an understanding of how a function such as Pascal's "WRITELN" prints data as output does not help a novice to see how to leave some vertical space between output lines. A structural appreciation of "WRITELN" views it as a combination of function and argument, which, when executed, causes the argument to be printed as a line of text. Novices must be taught the functional "trick" of calling "WRITELN" without an argument so as to produce a blank line because this "... potential function ... " is not clearly visible in the structural model.

This kind of "trick" is the stuff of "functional" models. They may tell us precisely how to achieve specific goals, but they do not describe the structure of the system with which we are interacting, and thus do not give us the information we need in order to make more general predictions about its behaviour. It is in these terms that di Sessa (1986) describes the characteristics of the two kinds of model, and their differing utility. The "structural" model is designed to offer explanation and correct predictions in uniform terms, and so focuses on characteristics which are independent of any specific use and applicable in all contexts. The "functional" model details characteristics which are concerned with "... specific use, consequences or intent"; (p. 202).

In the latter part of his paper di Sessa (op. cit.) sounds what must be a note of warning for any attempt to base a tutor's viewpoints on coherent mental models. He draws on work with 'Logo' to show that learners may not acquire a single functional 'frame' but a patchwork of partial explanations which are combined in what he describes as "distributed models". In addition to the stated functional model, these may include such elements as analogy to text, visual pattern matching, and rationalisations. Even if a coherent model is learned, it may be inconsistently applied.

2.2.2 The application of models.

di Sessa's point at the end of section 2.2.1 concerning the application of mental models raises an important issue for this discussion of viewpoints. Knowing a model does not guarantee its use, or indeed its use in appropriate situations. If the use of mental models may be characterised as problem-solving, then some significant and useful elements may be identified in the literature. This section discusses some of these.

In a much-referenced paper Larkin et al. (1980) attempt to discover what lies behind such words as 'judgement' and 'intuition' when attempting to explain the superior problem-solving abilities of experts doing physics problems. According to their analysis, experts

appear to use "... pattern-indexed schemata... " to relate significant features of the problem to the parts of long-term memory which retain the knowledge needed to find a solution. Put more simply, experts have a means of deciding what kind of problem they are dealing with, and thus what approach to it is appropriate.

A similar idea is advanced by Clancey (1985) who describes a problem-solving technique called 'heuristic classification' which may be successfully built into expert systems such as NEOMYCIN. The point of this technique is that it provides a heuristic which directly links two concepts in a non-hierarchical manner, and specifies the knowledge needed to solve a problem without reference to the specific programming language used to represent that knowledge. More standard classificatory methods rely on hierarchies to define the intermediate relations between the concepts and thus produce a more certain inference at the cost of more extensive search.

The fact that concepts are directly linked should not be taken to mean that the link between problem and solution are equally direct. Features of the problem are identified by data abstraction, a process which may have to go through several iterations before any of the features identified map onto parts of the solution schemas. This could mean, for example, that data relating to a specific patient is abstracted, initially, into categories which attempt to structure that data. The solutions themselves may be either selected from a known set or constructed in response to the problem features.

As Clancey points out, this technique has some strong implications for teaching systems which may be built to the heuristic classification model. He quotes Bruner and refers to others to show that the idea is not a new one but has previously been a significant part of educational, AI, and psychological theorising. We do not have to accept Clancey's system in all its detail to state some of these implications, but can take his (and Bruner's) point as to the importance of classification. One obvious conclusion is that knowledge of solutions needs some indexing or classificatory method to link it, even indirectly, with features in the

problem domain. This was also, in part, the message of the Larkin et al. (1980) paper discussed above. The necessity of such a link becomes all the more apparent if one is contemplating tutoring multiple viewpoints on a domain, as these viewpoints may relate to radically different forms of solution. The link is thus necessary to indicate the kinds of problem to which a viewpoint may be applied. We may again refer to Minsky's (1981) generator example (see section 2.1.7) to exemplify this: seeking mechanical solutions to electrical problems can be very unproductive. A more general consideration is that students may rapidly become disorientated if we tutor different viewpoints on a domain without indicating that these have different utility. This theme is taken up in chapter 3 where we formulate a structure for viewpoints.

A direct connection from the concerns of the preceding paragraph to the ITS literature may be found in such papers as Self (1989) and Collins Brown and Newman (1987). Self emphasises the need for 'reflection', ie. reasoning *about* our beliefs rather than with them, and reasoning about our problem-solving methods. The contention is that we may learn as much from reasoning about how we solved the problem, as from actually solving it. As Self points out, this could be a problem as our ITSs are not, yet, telepathic, and can not be aware of a student's reflection. A useful alternative to this is the separation of the 'task level' and 'discussion level' (Cumming and Self 1989b) for ITSs, where the purpose of the 'discussion level' is to critique the execution of the task. In terms of 'heuristic classification', this could mean that discussing the heuristics used could become a valid tutorial activity.

An example of such activity is given by Brown, Collins and Duguid (1989) under the banner of "cognitive apprenticeship". A college mathematics. class is given the "Magic Square" problem. The point of the activity is not simply for them to solve the problem, but more importantly for them to understand how mathematicians might think about the world, and set about solving such problems. This is achieved by having them "... enter the culture of mathematical practice" (ibid. p. 37), the idea being that profound learning can take place

through authentic activity akin to the training of an apprentice, rather than through the abstracted activity common to much schooling. The class worked through the problem collaboratively and then, crucially, went on to analyse the solution, thus highlighting the heuristics which led to the adoption of a particular solution method. Alternative solutions were analysed, and found to be examples of more general mathematical ideas. This "culture of mathematics" is contrasted with the "culture of schooling" whose practice would stop when the first solution to the problem was found.

2.2.3 Conclusions

This section has reviewed some of the literature on mental models which is relevant to our discussion of the nature and use of multiple viewpoints in ITS. Young (1983) has given us reasonable definitions of "surrogate", (structural), and "task-action mapping", (functional) models, while di Sessa (1986) has told us what each kind may be good for. The studies from Gentner and Gentner (1983) and Kieras and Bovair (1984) show that different inferences can arise when different (or no) models are used. Larkin et al. (1980) and Clancey (1985) have pointed out the importance of some form of indexing mechanism which links the problem to the knowledge which solves it. We show this to be relevant to the position of writers such as Self (1989) and Brown, Collins and Duguid (1989) concerning meta-cognition and learning. It is concluded that such linking mechanisms will be important for tutoring systems exploiting multiple viewpoints on the domain.

2.3 Prolog Tutoring and Tracing

The ideas being developed in this thesis are to be tested through the implementation of a tutoring system for a specific domain. The intended domain is that of debugging strategies for Prolog novices. This section reviews literature relevant to this domain. Bundy et al. (1985) provide a source of possible viewpoints, while Fung et al. (1987) and Taylor (1988) identify a range of novice misconceptions in Prolog which should be considered in the design of any diagnostic component for a Prolog tutoring system. Eisenstadt (1984,

1985) provides an implementation technique by which Prolog execution may be directly observed or interpreted. This technique can be adapted to allow the interpretation of Prolog execution in terms of our desired viewpoints.

2.3.1 Prolog novices have problems.

It is widely acknowledged (eg. Bundy et al. 1985) that Prolog can present problems for novices. It can be, for non-mathematicians, a complex and difficult language to learn. This section considers some of the literature on Prolog novices since it is they who constitute the target group of the proposed tutoring system.

It was recognised fairly early on that novice's had problems in visualising what was happening in the Prolog interpreter, and an early attempt to provide a consistent framework for interpreting the execution of programs was Byrd's (1980) 'box' model or "notional machine"; (similar in purpose to du Boulay et al.'s [1981] "glass box". See section 2.1.4). This is still the basis of many trace packages. It was helpful, but, as Pain and Bundy (1985) and Bundy et al. (1985) point out, it did not make clear the details of progressive resolutions, of outstanding goals, or of the search strategy being employed. The two papers just referred to also list other representations of Prolog execution, such as the "and/or tree", the "arrow diagram" and the "flow of satisfaction". None of these alone is seen to be adequate in isolation, since they emphasise different aspects of the execution, and do not all conform to a consistent model of the interpreter. Accordingly, Bundy et al. (1985) set out to produce a complete and consistent "Prolog story" which could be used to "...understand and predict the execution of a Prolog program", and which could form the basis for teaching materials, error messages, and tracing packages. The "story" was intended to cover both the procedural and declarative semantics of Prolog, and to illuminate the 'difficult to understand' aspects of the language such as the construction of recursive data structures and the scope of variables.

The resultant story has four parts. These are the Program Database, the Search Space, the Search Strategy, and the Resolution Process. In various combinations, these four models can be used to comprehend the many different representations of Prolog execution, (Byrd Boxes, Arrow Diagrams, And/Or Trees, Or Trees, Full Trace, Partial Trace etc.). The Database is the collection of assertion and implication clauses that go to make up the Prolog program. The Search Space describes the relationship between the goal literals which are input by the user, or generated by the program, and the Program Database. The Search Strategy is concerned with the order in which the goal literals are generated, and the order in which Database clauses are chosen for resolution with them. The Resolution Process describes the unification of goal and clausehead, and the possible generation of new goal literals. These stories are dealt with more fully in chapter 6 as they form the starting point for the viewpoints of the implemented tutoring system.

It is intended that the ideas developed in this thesis should be tested through the implementation of a system designed for tutoring novices in the domain of Prolog debugging strategies. It will thus probably be beneficial to consider just what problems and misconceptions such novices display. A range of these are detailed quite clearly in the work of Fung et al. (1987) and Taylor (1988).

Fung et al. propose an "initial taxonomy" of misconceptions found in novices' work with Prolog. Its categories refer to misconceptions about such areas as search and backtracking, backtracking alone, backtracking and the cut, the flow of control and variables, variables and the cut, and details of the unification process. These misconceptions may be simple, such as the idea that the interpreter always attempts to resolve a goal literal with facts before clauses regardless of their position in the database, to more complex conceptions such as the idea that variable instantiations to the left of a cut apply to the subgoals on the right of the cut, even if the variable names do not occur in that subgoal.

Taylor (1988) describes a range of 'superbugs' (Pea 1986) in addition to the lower level bugs identified by Fung et al. (op. cit.). These seem to be imported from the students knowledge of spoken or written language on the implicit assumption that Prolog will abide by the same rules. An example is the "left-to-right bias" superbug, which assumes that the execution will always move through the program clauses in the manner that we read text, that is top-down and left-to-right. This may occur some of the time, but not all of the time.

These descriptions of novices' misconceptions do not seem to have any immediate implications for our first task, that is to develop a structure for implementing viewpoints, and to design a tutoring system which may use them effectively. If, however, that system is to contain a diagnostic component which needs to take account of any misconceptions which the student may display, then the work of Fung et al. (1987) and Taylor (1988) immediately becomes more relevant. Initially, more attention will be paid to the work of Bundy et al. (1985) and the four elements of their Prolog "story".

2.3.2 A technique for interpreting a record of Prolog execution.

The problems that people exhibit in understanding Prolog programs has led to another research direction. This aims to produce better tools for tracing and debugging Prolog programs, and caters for practitioners as well as novices. A survey of available tools is given in Brna et al. (1988) who list the types of tool and the Prolog environments which support them. The desire to improve on the 'boxes' model of execution which underlay most tracers (Byrd 1980), led Eisenstadt (1984, 1985) to develop PTP, the "Prolog Trace Package". This uses 19 symbols to describe an execution trace at a fine grain level. In particular, the symbols reveal the detailed symptoms of all resolution attempts, highlighting such events as subgoal success and arity failure. The symbolic trace is stored with the relevant goal and clause for each line. This information is used to provide different levels of detail for the user of the trace. They could, for instance, determine the end result, and

then "zoom" in to the appropriate level of detail in the execution. PTP also uses the trace to look for "bug cliches" in the execution.

The interest of this technique in relation to our thesis is that it provides a means of interpreting Prolog execution after the program has been run. If the technique can be used to interpret Prolog execution in terms of the four-part "story" proposed by Bundy et al. (1985) then we may have the possibility of implementing viewpoints based on the "story", and a tutoring system based on the viewpoints.

2.3.3 Conclusions: Prolog as an implementation domain

In section 2.3 three areas of research are reviewed which, taken together, indicate that some aspects of novice-level Prolog may be a fruitful area in which to implement an ITS incorporating multiple viewpoints. Bundy et al.'s (1985) "story" shows the need for complementary models of Prolog which highlight different aspects of the language's execution. It is assumed that these "stories" could form the basis of viewpoints in the proposed tutoring system. Fung et al. (1987) and Taylor (1988) identify a range of misconceptions which the design of any diagnostic component for a tutoring system would need to consider, the system being proposed in this chapter being no exception. Eisenstadt (1984, 1985) provides a technique which can be used to build the interpreter that the proposed tutor, with its different viewpoints, would require.

2.4 Debugging: studies and tutoring

Debugging is a rich and complex topic, with many levels of analysis from program syntax to programmer semantics. The focus of this research however, is not debugging per se., but the tutoring of debugging strategies. This section describes how a limited and simplified catalogue of bugs can be identified in relation to an ideal version of the code through the work of Brna et al. (1987). The proposed system is contrasted with others such as PROUST (Johnson and Soloway 1984), and the point made that it is not intended

to constitute a 'debugger' in the conventional sense of the word. The intended research direction is to tutor the application of appropriate models rather than to build a full-scale debugger. The simplification of the 'bug' domain means that the research efforts can be concentrated on developing mechanisms which describe the bug's effect on execution in terms of the four models we wish to use, and which are able to tutor this skill.

2.4.1 Describing Bugs

The four parts of the "story" outlined in Bundy et al. (1985) could be applied to Prolog in many ways, (eg. to describe general execution as opposed to specific programming techniques or algorithms). In order to focus the design problem we have decided to concentrate on the area of debugging for novices.

This presents the problem of how we are to describe and classify bugs in a manner suitable to our purposes. Help is at hand in this matter from Brna et al. (1987). This paper provides a method of classifying bugs based on programmer expectations. Brna et al. define four levels of description for a bug. These are the 'Symptom Description' level, (eg. a wrong variable binding); the 'Program Misbehaviour Description', (the explanation offered for a symptom in terms of control flow); the 'Program Code Error Description', (eg. abstractions such as a 'missing base case'); the 'Underlying Misconception Description'. As the proposed system is aimed at novices, it will require a simple system for classifying bugs. Our immediate concern is thus with the 'Symptom Description', and the 'Program Code Error Description' levels, as it is assumed that novices are not well-equiped to deal with descriptions of control flow and misconceptions. Apart from error messages and side effects, symptoms may concern Termination issues or the Instantiation of Variables.

If we concentrate on the Variable Instantiation bugs, these may be classified as

a) the unexpected failure to instantiate a variable;

b) the unexpected instantiation of a variable;

c) a variable instantiated to an unexpected value.

These three behaviours may be explained at the 'Code Error' level in terms of 'modules'. Depending on the level of description, these 'modules' may be such entities as a set of predicates, or within a clause, an argument or subgoal. A given fault may be due to a missing module, an extra module, or a wrong module; (eg a missing subgoal, an extra subgoal, or a wrong subgoal). The search strategy of Prolog requires that we also include the possibility of wrong order for clauses and subgoals. If our list of modules is complete, then all possible (individual) bugs may be described in this way, in relation to some ideal "template" code. Although this classification may be described as 'syntactic' in that it does not include any of the procedural semantics of the programmer, it has the advantages of being simple, regular, and complete, while defining a finite number of bug types. Using this classification, we would be able to specify exactly what bugs any prospective system would be able to process, without reference to the intentions of the programmer.

The classifications of the 'Program Misbehaviour Description' become relevant once the domain has been formulated, and possible dialogues between system and student are being considered.

2.4.2 Debugging Systems

The ability to classify bugs without reference to the programmer's intentions is of crucial importance. The classification outlined in 2.4.1 refers the bugged code and result to an ideal version of the code and its result. Any system operating in this manner would thus be quite distinct from a "debugger" which attempts to find bugs in an arbitrary piece of code. The debugger has to try and cope with such issues as the different possible ways of solving a given problem, ie. there may be many different assemblages of code which give the same result. Another issue for a debugger is that of acquiring some representation of just what the programmer was attempting to do in the first place. If the program syntax is

correct, then any bugs must lie in the relationship between what was intended and the manner in which the program actually executes. We now briefly describe two systems which *do* utilise some representation of the programmers intentions, to illustrate how these systems might differ from one which utilises an ideal version of the code.

PROUST (Johnson and Soloway 1984) was intended to be able to diagnose bugs in the code of students who were given a specific problem. The system attempts to analyse a formalised version of the problem and the students solution by synthesising the design process that led to the student's code. The exact set of intentions which gave rise to the code were not available to the system, but had to be inferred by it on the basis of the problem specifications and the input code. PROUST used a three-level mechanism for this inferencing, consisting of *goals* and *subgoals* selected on the basis of the problem specification, *plans* to realise the *goals*, and *code* to implement the *plans*. The system had detailed knowledge of each level, and various forms of rules for dealing with mis-matches. The detection of bugs has to proceed in parallel with the synthesis of the programmer's intentions as bugs might lead to a misinterpretation of the intentions, and a representation of intentions was needed to spot bugs. This may be likened to a form of parsing. PROUST was quite successful in identifying bugs from simpler problems, but more complex problems allow the programmers greater freedom, and require more bottom-up analysis to infer intentions directly from the code. This is very hard and Johnson (1987b) was led to consider ways in which programmers might explicitly discuss their intentions with the diagnosing system.

This method of explicit discussion between the system and the programmer was actually used in SNIFFER (Shapiro 1981). This system stored a complete record of the code's execution, and had a number of tools which allowed the programmer to interrogate this record in order to locate the point where unexpected behaviour was obtained. The programmer then gave this point and a statement of what *was* expected as input to the system's debugging experts. These compared the actual code associated with the indicated

region with the code that the expectation would require in order to determine what bugs may be relevant.

2.4.3 Conclusions: Prolog debugging as an implementation domain.

It should be clear from the remarks in 2.4.2 that the distinctions captured by the "symptom" and "code error level" sections of Brna et al.'s (1987) paper do not support the construction of a 'debugger' as the term is generally used. Nor do we wish them to, since that is not what we wish to build. In general terms our goals are as follows: we wish to investigate whether the four models of Prolog execution outlined in Bundy et al. (1985) can be implemented as viewpoints in a tutoring system which focuses on the use of those viewpoints in debugging. In other words, we require a system which can tutor the skill of using the four models of Prolog execution to localise bugs in Prolog code. This does not necessarily require the high-level intention-recognition and bug-recognition facilities of PROUST (Johnson and Soloway 1984), as we may limit the definition of what a bug may be, and instead put the effort into a system which can describe those bugs we do allow in terms of the four models of Prolog, and tutor in relation to this skill. Our research direction is to tutor the application of the models rather than build a debugger. Brna et al.'s (1987) paper gives us a complete classification of bugs in relation to an ideal version of the code. It is concerned only with the 'ideal' and 'bugged' results, and not with any question of intention.

For such a closed and structured world, a 'bugfinder' should not be difficult to build, and efforts could be concentrated on developing mechanisms which describe the bug's effect on execution in terms of the four models we wish to use, and which are able to tutor the skill of using those models to localise bugs. In short, we wish to demonstrate that a system may be built which can support a satisfactory tutorial dialogue with the student in terms of different viewpoints on Prolog execution, and which can help the student to use those viewpoints effectively in localising bugs.

2.5 Explanation Content and Knowledge Base Structure.

Much of the effectiveness of any tutoring system depends on the quality of explanation that it is able to give. It was this issue which prompted the inclusion of 'meta-rules' in NEOMYCIN. (see section 2.1). As the issue is certain to arise in the implementation and use of the proposed system, a brief review of explanation in relation to multiple viewpoints is included here. The major concern is to establish how the content of explanations may vary with the viewpoints they relate to. There appear to be two methods by which such adaptive explanations may be produced: either the distinct viewpoints may be built into the knowledge representations of the system, or else a mechanism may be implemented to control what is retrieved from previously-existing knowledge representations which are not dedicated to a specific viewpoint. This section discusses systems which illustrate the use of both these methods.

2.5.1 Structuring the Knowledge Base.

Stevens and Steinberg (1981) describe "A typology of explanations and its application to computer aided instruction". This work was an offshoot of the research on STEAMER (Hollan et al. 1984, see section 2.1), and drew on naval engineering texts and operations manuals to produce a taxonomy that could be used for organising explanations of physical devices. They distinguish nine types of explanation, which differ in

> "... level of detail, conceptual perspective, degree of 'match' with physical reality, and degree of 'quantitativeness'" (Stevens and Steinberg 1981 p. 2).

The "physical-causal" explanation for example, uses words such as 'push', 'pull' and 'force' to break a continuous process into sets of discrete events which have causal links between them and are temporally ordered. The "stuff-state-attribute" explanation on the other hand emphasises the substances that the system in question is dealing with, and the way in which the attributes of the "stuff" change with a change of state. It exploits the

strong human perception that "stuff" is conserved, (what goes in must come out, somewhere), to assist inferences about such matters as changing levels in water tanks.

Stevens and Steinberg (1981) claim to have automated the "physical-causal" explanation and conclude that multiple types of explanation are needed precisely because people use several different types of mental models to reason about complex physical systems. Thus, they maintain, the goal for a tutor is to:

> "... determine what models are necessary to reason in the ways useful to perform the tasks associated with a given system and then communicate explanations embodying those models". (Stevens and Steinberg 1981 p. 18).

An example of a system which has achieved this in some degree is the 'Recovery Boiler Tutor' (Woolf 1988). This system tutors the operation of a complex piece of plant found in many paper mills throughout the USA. It adjusts its explanations to different operational situations and to different operators. The system combines a simulation with an intelligent tutor and monitors the operator's performance. The operator can request information about the state of the system in many forms, eg. as an animated graphic depicting the system's components, as a control panel of gauges, or as specific trends plotted against time. Via menus the operator may ask for information about such matters as the current problem, the appropriate actions, and the root cause of the situation. The explanations given in response draw on a domain representation which is "broken" into three classifications. The *'conceptual knowledge'* identifies the basic domain concepts, and the relationship between them. *'Procedural knowledge'* encodes the reasoning used by the system to solve problems in the domain. Exploiting both of the former is the *'heuristic knowledge'* which is intended to describe how an expert would make measurements in the domain, and manipulate the *'conceptual'* and *'procedural'* knowledge so as to find solutions. It could be argued that these different forms of knowledge constitute the different 'models' that Stevens and Steinberg (1981) wish to see as the basis of tutoring system explanations.

The 'Recovery Boiler Tutor' also adapts its explanations at an individual level in relation to actions taken by a trainee. When an operator has taken some action in response to an undesirable situation, the system can offer a critique of that action. The system refers to a set of *'scenarios'* which detail preconditions, operating emergencies, their solutions and post conditions, to evaluate the student's action in the current *'scenario'*. They may, for instance, be told that the action was 'safe but not optimal', and why this is so. Woolf (1988) stresses how the system's domain representations were carefully structured to provide these different person-related and system-related forms of explanation.

Partitioning of the knowledge base is a technique also used by McKeown (1988) for the ADVISOR system. This was not a tutoring system, but was intended to provide advice to students about which courses they could or should take. The system interacts via a natural language interface and the user's goals are derived from the ongoing discourse. With the goal established, an underlying expert system determines the answer for this student, and an edited version of the execution trace of this system is used to generate the explanation. The knowledge base used for this is partitioned in terms of "perspectives", and the possible goals are related to these perspectives in such a manner that the same advice may, in different conditions, be justified by different explanations. The content of these explanations is chiefly influenced by the user's overall goal, eg. 'how can I fulfil requirements as quickly as possible?' as opposed to 'how can I maximise my personal interest?' Relevant information about these goals is included in the explanation.

The different perspectives are represented by hierarchies of data whose roots are such concepts as 'topics' (eg. 'AI topics') or 'requirements' (eg. the prerequisite credits for doing a course in data structures). These hierarchies are linked by entities such as courses which may be viewed from more than one perspective. The relevant information is retrieved from the hierarchy and used as input to the expert system, and the resulting trace edited into the explanation. A notable point is that a single hierarchy may be accessed by more than one goal, so that different information is extracted, and a different trace

produced. This may be compared to SOPHIE I, (Brown and Burton 1975), where a single circuit representation is accessed in terms of several different viewpoints.

2.5.2 Interpreting the Knowledge Base.

A different technique is advocated by Souther et al. (1989) and Suthers (1988). Souther et al. are concerned with the provision of explanation in relation to the foundational knowledge of introductory college courses. A feature of these courses is that they draw on a large number of highly interelated viewpoints. Souther et al. note that previous designs for ITS usually relate to a much more restricted domain, and the reader may indeed express surprise at the scope of Souther et al.'s ambitions. They propose a method which utilises

> "... domain-independent knowledge in the form of *view types* to select the appropriate knowledge "

from a generalised knowledge base. (Souther et. al 1989 p. 123).

Elements of the domain knowledge are provided with annotations which indicate when that element should be included in a certain type of explanation. The goal is thus to generate explanations dynamically in terms of a small number of "view types". These "types" are each intended to deal with a specific category of questions that a student might ask, and each has a "strategy" associated with it which determines how that view is applied to the knowledge base to generate an explanation. In addition, a view type specifies "necessary" relations, which must be included in the explanations, and "permissible" ones, which may be included but are not *required.*

The view types have such labels as "functional", "modulatory", "structural", "class-dependent", "attributional" and "comparative". (These seem to overlap to some degree with the explanation categories provided by Stevens and Steinberg [1981], as when considering the attributes of objects). The explanation is based on a "concept of interest"

which, for certain "view types" is related to a "reference concept" to generate the required information; (eg. pollen in relation to plant reproduction). The "strategies" select domain knowledge in relation to the question asked and the reference concept.

The authors give us examples of the "definition-generation" strategy applied to three view-types for the question "what is photosynthesis?", and claim to have produced explanation generation "strategies" for two classes of questions, "... definition requests and comparison questions" in relation to six view types. These strategies were able to generate fifty test definitions from a botany text book by selecting links in a semantic net.

What began as an attempt to simplify the "intractable" problem of explicitly representing all relevant viewpoints in a knowledge base begins itself to look highly complex here. There are some seven view types, each with a number of different explanation strategies which must take account of 'necessary' and 'possible' relations to a very large number of reference concepts, and which must be able to process and present temporal, spatial, taxonomic and teleological information. Does this method imply that each element of the domain knowledge base is to be indexed in relation to all of these categories? Such a task does not seem tractable for a knowledge base of any size. This work could be characterised as an attempt to erect a powerful epistemology on a conceptually simple foundation, and in this sense its wisdom could be compared with that of similar attempts under the name of "conceptual dependency" (Schank 1973). We may also wish to question the extent to which a "domain independent" taxonomy of viewpoints may be defined.

Other questions may be asked in relation to the educational benefit of the exercise. Souther et al. briefly mention the use of a student model and a dialogue history to provide "context-specific presentations", but is education simply a matter of providing relevant definitions? If these can be found equally well in a text book, (the comparison they themselves chose), what is the benefit of using the system? There appears to be a distinct qualitative difference between systems which actively engage the student in problem-solving activities and

critique their performance, such as SOPHIE (Brown and Burton 1975), WEST (Burton and Brown 1979), or the Recovery Boiler Tutor (Woolf 1988), and those which simply present information to the student in the manner of a book, (albeit with more adaptation to the form of the student's question than a book can provide).

The explanation mechanism proposed by Suthers (1988) is not so overtly dedicated to tutorial use, (although its use in this context is mentioned in passing), but has a comparable approach. His basic idea is to free the discourse or explanation manager from having to conform to the structures of a knowledge base which is structured mainly for reaching decisions rather than for justifying them. This is to be achieved by using a "view retriever" which accesses the knowledge base and provides a suitable form of input to the discourse manager. These "views" are retrieved "... by parameters given along a small set of epistemological dimensions" (Suthers 1988 p. 436). The "Topic" dimension defines the object of central interest, and in the example Suthers gives us, (psychoeducational case analysis), has two sub-dimensions "type" and "generality". Just how "topics" and "sub-dimensions" are to be specified in the domain independent terms desired is not made clear.

The "model" dimension determines the conceptual framework which is to be applied to "... a given instance of the machine's reasoning", and contains the sub-dimensions "domain", "computational", and "implementational". The last two of these deal with abstractions about AI and the architecture of the reasoning machine. The "domain" sub-dimension deals with the alternative models applied by practicioners in the field, and appears to assume that all these models are equally applicable to a given instance of the machine's reasoning, and that the machine's knowledge representations are rich enough to support them all. The last dimension is that of "organisation", and specifies the way in which the explanation should be structured, ie. what relations it should contain. Suthers lists six "classes" of relations (eg. "chronological" or "structural"), but there seems little reason in principle why this list of classes could not be extended indefinitely.

To retrieve a specific view, the specifications for each of these dimensions need to be backed up by a statement of the level of detail required, and a set of tools which map the elements of the knowledge base. In terms of a semantic net, these would be "node tables" and "arc tables" which constrain the construction of a view in relation to the "epistemological dimensions", and "path grammars" which constrain which nodes in the representation may be traversed. These tools are not described in detail, but are intended to encode the bulk of the domain-specific data. They appear to be related to the "semantic grammar" of Burton (1975, 1976), and must represent a serious investment of effort, since they must be constructed in relation to each dimension and sub-dimension. An implementation is said to be in progress.

2.5.3 Conclusions.

We may make a general point about the papers and systems considered in this section, (2.5). They all envisage or describe explanation mechanisms which utilise pre-defined knowledge representations to produce explanations whose contents vary with the goals of the user. If we choose to see these different knowledge representations as encoding 'viewpoints', then our goal of designing a tutor that is based on pre-defined viewpoints and that can produce appropriate explanations in relation to them begins to look realisable. The implementations of 2.5.1 rely on knowledge representations which are purpose-built for explanation, advice, or tutoring. This greatly simplifies the extraction and processing of the relevant information, and gives greater scope for tutorially useful interaction.

The systems discussed in 2.5.2 rely on conceptual structures which interpret knowledge bases designed for problem-solving performance. These structures are intended to be simple, but inevitably become much more complex, while their implementational value has yet to be proved and their educational value may well be questioned.

Where structures are defined for viewpoints, (Souther et al. 1989 and Suthers 1988) they appear to be largely taxonomic, and do not focus on the issue of how the knowledge is to be used.

2.6 Educational Philosophy.

Every tutoring system, intelligent or otherwise, implies the existence of some educational philosophy, explicit or otherwise. This author is of the opinion that good design is more likely to be achieved where a clear and explicit educational philosophy is available to guide the process. At the very least, it helps to set explicit educational goals so that the success of the system may be judged, and to remind the system builder that their creation has a purpose beyond technical wizardry. This section sets out the educational philosophy which underpins the research.

2.6.1 Ways of learning.

> " Good learning situations and successful ITS, I suggest, are successful not because they enable a learner to ingest preformed knowledge in some optimal way, but rather because they provide initially underdetermined, threadbare concepts to which, through conversation, negotiation, and authentic activity, a learner adds texture". (Brown 1989 p. 4).

This quotation exemplifies an emphasis on 'authentic activity' which pervades the recent writings of many who are concerned with "situated cognition" such as Brown (1989). Brown refers to Resnick (1987) to distinguish the different kinds of learning that take place in and out of school, and argues that the abstracted, explicit and generalised knowledge that is the focus of formal teaching is not only difficult to learn, but is only marginally transferable. He maintains that when it is situated in actual activity, the cognition of 'just plain folks' or 'JPFs' (Lave 1988b) has much in common with that of experts, and that any attempt to promote learning should take account of this. As Brown draws specific

conclusions from this in relation to ITS design, and as he summarises the work of many others in this area, we shall examine his argument in more detail.

Brown seeks to distinguish between two views of learning. The conventional or 'didactic' view emphasises "... explicit renderings of "knowledge" ..." which are decontextualised and which promote problem-solving as the central quality of expertise. Brown suggests a different view, derived from studies of ordinary people engaged in activities from which they learn. This view stresses that learning is a process of making sense of the world in a social and practical context, where learners produce, or co-produce concepts and models in response to both the activity and some minimal concept specification. Brown supports this view by contrasting learning in and out of school in terms of four categories noted by Resnick (1987).

The first of these refers to authors such as Lave (1988a) to show that, outside of school, most human activity, including learning, is social in nature, and that knowledge itself can be seen as very much a social construction. While people may learn communally, they are generally *taught* individually. Resnick's second category describes "... pure mentation... " as opposed to "... tool manipulation." Schools emphasise the former, to the point of insisting that "props" allowed in learning should not be available during testing. This ignores the fact that tools constitute a large part of our environment, and that people use apparently unrelated parts of their surroundings to distribute the burden of cognition. An example of this given later is Lave's (1988b) account of a shopper finding the cheaper piece of cheese in a bin which contained two sorts of the product, but gave no unit price. Rather than engaging in complex calculations to determine the unit price, the shopper simply found two pieces of equal size and selected the cheapest.

The third category discussed by Brown compares reasoning about abstractions with reasoning about "stuff" (*sic.*). Outside the schools, people usually manipulate the "stuff" of the world directly, or else manipulate abstract symbols which are very closely tied to

such "stuff". These ties allow an expert to "look into" the world through the abstractions, as an architect sees a building through a drawing or a physicist sees a circuit through a diagram, so that the abstractions themselves are 'transparent'. School work, on the contrary, emphasises the manipulation of symbols which have few connections to the world, so that they become an end in themselves, and are thus "... opaque and disconnected..." rather than a means to an end.

The last category deals with generalised learning, as opposed to that which is situation-specific. Most school learning is deliberately abstract and generalised, with the intention that it should be easily transferable. It turns out not to be so, as well as being difficult to learn. Brown cites several authors on this point, and it is also supported by Cowan (1986). Students are frequently unable to apply classroom knowledge except on classroom tests. Brown suggests that situation-specific learning is however transferable, through processes of analogical "intuition".

Having established these differences, Brown goes on to consider the main characteristics of "Everyday Cognition" and "Expert Cognition". The former typically uses elements in the environment to bear some of the load of computation and representation as with the 'cheese' example given earlier. The point being made is that people, rather than solving problems outside of the context in which they encountered them, seem particularly good at solving them *within* that context. Experts too exploit the context in their reasoning in implicit ways, seeming to use contextual cues to decide what kind of analysis (or viewpoint?) to apply to a given problem. Depending on the assumptions that an expert makes, a transistor can be "seen as" an amplifier or a switch, and the implications of this explored in causal terms (Brown and Burton 1987). It is this exploration which connects the abstraction to the reality. These remarks seem to have much in common with the ideas discussed earlier in relation to Clancey's (1985) "Heuristic Classification", where a sequence of classifications connects the problem and the knowledge required to determine a solution.

These perceptions lead Brown (1989) to argue that what JPFs do, and what experts do is quite distinct from what is done in schools, and that there is a surprising similarity between the "... implicit reasoning *processes* ..." (*sic.*) of JPFs and experts. There is thus a real continuity between the cognition of JPFs and that of experts, while there is a real discontinuity between the cognition of school students and experts. These continuities and discontinuities are detailed, and two summary points are made: the activities of JPFs and experts are situated within cultures of activity which strongly influence the negotiation of meaning between situation and actor, and the construction of understanding, (see remarks on Brown, Collins and Duguid [1989] above); in contrast to this, students are generally expected to work on de-contextualised symbols and laws applied to well-defined problems which are not related to any interpretive culture.

Brown concludes that the key implication of all this for ITS is that we must design systems to do what schools have not done: we must take advantage of the "... robust, innovative features ..." of human learning that he has described. This implies that we must re-assess our ideas of what constitutes educational practice so as to include both formal and informal learning of both explicit and inexplicit knowledge. Brown refers to studies such as Schon (1987) to point out that an account of expertise which relies solely on the formal "... abstractable content ..." as a basis for learning is not going to be an adequate basis for producing experts. Experts inhabit a "... practice world ..." of

> "... conventions, constraints, languages, and appreciative systems, ... repertoire of exemplars, systematic knowledge, and patterns of knowing-in-action." (Schon 1987 pp. 36-37).

It is the process of enculturation into this world that is referred to in Brown, Collins and Duguid 1989 as "cognitive apprenticeship". The challenge for ITS is thus not to produce successions of microworlds, but "Increasingly Complex Enculturating Environments".

Brown goes on to list five crucial aspects of learning which he believes are currently overlooked, and which must be taken account of in the design of the "Enculturating Environments". At the core of these is a demand that the technological and conceptual tools that constitute the ITS must be "transparent" in the sense that they are seen as the means to an end, rather than an end in themselves. The students must be able to "see through" the tools to the world itself as a blind person can appreciate the world through their cane without significant awareness of the cane itself. The attempt to characterise this kind of technology gives rise to a rather different use of the term "Glass-box" from that used in section 2.1.4. Brown's use of the term goes beyond identifying knowledge representations which are adapted to the provision of tutoring and explanation, to a more demanding design imperative which stipulates that the system and its knowledge representations should connect the student directly to the world of practice. This may sound like a contradiction in terms since an ITS can only ever constitute a virtual world. The stipulation takes on more meaning though, if it is interpreted as a demand that the use and practice of the knowledge to be learned should always have a central place in the ITS design process.

Brown finally identifies three types of "transparency" that a system may manifest, and refers to Wenger (1988) for a more comprehensive taxonomy. "Domain transparency" describes qualities in the learning tool that allow the learner to "see into" the domain, focusing on those aspects of it which are of interest without being distracted or obstructed by other features of the tool itself. The analogy is drawn with the use of a magnifying glass. The glass is transparent, but the object is brought sharply into focus. This may be interpreted as offering models of the domain which are suited to the learner's purpose.

"Internal transparency" refers to qualities of the learning tool itself. Where a tool or system aids performance in some area through built-in expertise, the reasoning strategies of the tool should be made clear so that the users may build for themselves suitable models of

successful reasoning in that domain. This sounds like the interpretation of "glass-box" given in section 2.1.4.

The third form of transparency is not so clearly described. "Embedding transparency" refers to the overall process of which the tool and its use are a part. The tool itself must not become "decontextualised", since then it becomes simply another rigid abstraction. Instead, the design of the tool must reflect the process of ongoing interactions between the user and the world. The implications of this for ITS design are not spelt out.

The goals articulated by Brown (1989) can be related to our discussion of viewpoints. "Domain transparency" can be taken as referring to the selection and use of appropriate domain models. If different models each have a different utility, as discussed above, then in order to provide the "transparency" that Brown advocates, the system will have to select an appropriate domain model for the student's goals. This in turn implies that the different models, and conventions regarding their use, will have to be represented in the system. This is indeed the project which we have outlined above.

The notion of "Internal transparency" raises the question of the degree to which the inference mechanisms that operate on the system's model can be made to match those of experts in the world. This issue, and that of the degree to which such a match can promote implicit learning, are matters which are better discussed in relation to the implementation described in subsequent chapters. The closeness of this 'matching' will also have implications for any consideration of the system's interface, as the development of STEAMER (Hollan et al. 1984) demonstrates. This issue is also best discussed in relation to the implemented system.

2.6.2 Applying Knowledge.

We interpret Brown's (1989) demand that a learning tool should always connect the learner to the world of practice as a demand that the use and practice of the knowledge to be learned should always have a central place in the ITS design process.

This is, after all, one of the things which distinguishes it from the explicit and formalised learning common in schoolwork with its apparent lack of transferability. These points about use and transferability are supported by Cowan (1986) who laments the inability of many students to apply the algorithms they had learned in engineering courses to problems whose nature was not immediately clear, ie. they were not good at analysing the problem to determine which algorithms should be applied. Cowan puts much of the blame for this on the manner in which the courses are taught: problems set are generally straightforward applications of algorithms which have just been tutored, and become ends in themselves rather than being intimately tied to engineering practice. As noted above, Cumming and Self (1989b) advocate an emphasis on such practice in terms of a "discussion" level which reflects upon the use of knowledge in problem solving. The purpose of the 'discussion level' is to critique the student's execution of the task so as to make them aware of possible alternatives or improvements.

This is the kind of separation described by Brown, Collins and Duguid (1989) in relation to the maths class which did not simply solve a problem, but spent a great deal of time subsequently discussing different possible solutions. In the light of Brown's (1989) remarks we may speculate that much of the effectiveness of the 'situated' learning that these authors describe comes from the fact that, being learned in practice and within a 'culture', the knowledge and its application are, from the first, intimately connected. The examples given by Stevens, Collins and Goldin (1979) indicate that integration between different viewpoints is as important as knowing in how a specific one should be used. We may finally refer once more to Larkin et al. (1980) who stress the expert's skill in analysing a

problem, and to Clancey (1985) who stresses the importance of "heuristic classification" as a means of connecting knowledge with solution strategies. The real point being made here is that our concern with the *application* of viewpoints is not an implementational convenience, but an issue of considerable importance for education itself.

2.7 Conclusions to Chapter 2.

Viewpoints appear to be alternative modes of analysis which each highlight different aspects of the domain in question and thus reduce the search space for problem solving. The review of ITS literature shows that different viewpoints may be needed both for tutoring and for problem-solving. There are indications that students learning different viewpoints on a domain also need to learn in what context the viewpoints are to be used, and how they are related to each other. Different viewpoints may also relate to different levels of expertise.

For an implemented ITS the knowledge representations of a large domain will probably need a degree of 'psychological plausibility' while smaller domain may be well served by a 'black-box/glass-box' combination. The basic problem seems to be that of finding a suitable conceptual structure for a viewpoint on which to base an implementation, and of deciding what educational goals and design philosophy it is to support. Where a range of viewpoints are available, the student may be seen as an active partner in the process, so that their learning history and goals may influence the selection of the viewpoint to be tutored. (This selection could be made by the student, a human teacher, an ITS, or any combination of these three).

The literature on mental models is taken as a reasonable starting point in looking for definitions of 'viewpoints'. Parts of this literature indicate that different inferences are made with different models, and that having a model may give better performance than having none. The nature and utility of 'structural' and 'functional' models are described in detail. The importance of mechanisms which link problem contexts to the knowledge

required for their solution is underlined, and a candidate mechanism offered by Clancey (1985).

A suitable domain for implementing an ITS appears to be that of Prolog for novices. The complementary models required, and common misconceptions about Prolog, have already been described by others (Bundy et al. 1985, Fung et al. 1987). An implementation technique which allows the interpretation of a record of Prolog execution has also been described (Eisenstadt 1985). The desire to emphasise the *application* of viewpoints leads us to consider the application of the models of Prolog to debugging as a domain. A means of describing bugs in terms of symptoms and program code (where 'template' code is available) is provided by Brna et al. (1987) . Relating possible bugs to template code allows the construction of a closed domain and avoids the complications inherent in building a 'debugger'. Implementation efforts may thus be concentrated on mechanisms which can describe and tutor the effects of particular bugs on Prolog execution in terms of Bundy et al.'s (1985) four models.

Part of a tutor's function is to provide appropriate explanation. Previous work in providing explanation in relation to different viewpoints is encouraging in that some success has been achieved using pre-defined viewpoints for specific domains (Stevens and Steinberg 1981, Woolf 1988, McKeown 1988). The proven implementations have structured their knowledge representations in terms of the relevant viewpoints. Another approach attempts to re-interpret knowledge-bases specifically built for problem-solving in the domain (Souther et al 1989). This approach appears to involve a highly complex conceptual apparatus, has yet to be implemented convincingly, and is of questionable educational value.

The use of multiple viewpoints in tutoring systems appears to entail some re-thinking of our educational philosophy since the application of each viewpoint also needs to be tutored, as does its relation to other viewpoints. This contextualises the knowledge to be learned in

a way that traditional schoolwork generally avoids, and provides the possibility of adapting our tutoring to the goals and experience of the student. An approach to education which emphasises the context and use of knowledge, and the situated nature of learning is proposed in Brown (1989), and is summarised in section 2.6. The implications of this for ITS design, as listed by Brown, are also summarised.

The goal of this thesis is thus to identify and test a structure for implementing viewpoints in tutoring systems, and to investigate the design issues which arise when several such implemented viewpoints are to be used in a single system. These design issues include both technical questions relating to system architecture and knowledge representation, and more general educational questions relating to educational philosophy, tutorial strategies and goals, and the provision of explanation.

Chapter 3. A Formulation for Viewpoints.

This chapter considers how the notion of a mental model may be augmented so as to provide a structure for representing viewpoints in an ITS. (This does not imply that the resulting viewpoint is proposed as a 'psychological reality'). The need for inference mechanisms to act upon a formalised model is considered, as is Wenger's (1987) attempt to formulate a viewpoint structure. The structure advocated in this thesis is developed by considering issues relating to the domain being tutored, the design objectives stated for the implemented system, and the educational philosophy it is intended to embody. As described the structure of each viewpoint is intended to be modular, consisting of a model for the domain, a set of inference procedures to act on the model, and a set of heuristics stating the application of the model/inference procedure combination to problem contexts. This formulation is considered in relation to the 'cognitive apprenticeship' theory of learning, and also in relation to alternative possible formulations. The need for a study to extend and validate the formulation is considered.

3.1 Considerations for the formulation.

What structure should we use to describe viewpoints? The literature considered in chapter 2 section 2 indicated that research on 'mental models' may have much to offer as a starting point for any formulation of 'viewpoints' for an ITS implementation. The point was made that to formalise the subjects' performance in the Gentner and Gentner (1983) study, and the Kieras and Bovair (1984) study in terms of the notion of viewpoints outlined above, (ie. to provide a formalisation which could act as the basis of a performance simulation for tutoring purposes), we would need to augment the studies in several ways.

These augmentations would need to include:

a) some account of the means by which inferences are drawn from the models;

b) some account of how a particular model is chosen as being applicable to a

particular problem;

c) a formalised version of the models used.

(Strictly speaking, the subjects of the two studies were *instructed* to apply certain models, but as we have argued above, knowledge about when to apply a model is crucial to its successful use). Since it seems likely that the final application of the models would depend to some extent on the inference processes that are used in conjunction with them, it seems logical to focus initially on the first and last of these questions, ie. the inference procedures and the model formalisation.

Although this discussion refers to 'mental models', it should be stressed that the concept of a "viewpoint" is not proposed as a psychological reality. We are concerned with the design of Intelligent Tutoring Systems, and a "viewpoint" is proposed only as a concept which is useful in this practice. The problem being addressed is that of how tutoring systems may be designed to utilise different viewpoints on a given domain. This stance is taken since the work of system design requires a performance simulation of appropriate reasoning in the domain, while psychologists continue to debate the psychological reality of their various accounts. Viewpoints are thus conceptualised as being based on 'models', and related heuristically to contexts of use. 'Model' is used here, and throughout this thesis, as by Young (1983) to describe a 'user's conceptual model'.

Viewpoints are seen as being distinct from available accounts of mental models in that firstly, they propose some mechanism for deciding when the model is applicable; and secondly, they propose specific mechanisms for making inferences from the model, and for choosing amongst these mechanisms. In general terms, then, mental models have to be applied to problems, and as we wish to describe this process for the implementation of tutoring systems. A viewpoint is thus seen as a description of the application of a mental model.a viewpoint is seen as a description of the application of a mental model. Its

purpose, loosely stated, is to allow the system to conduct its tutoring flexibly, taking into account the goals and knowledge states of the student.

The issue of how different inference procedures may be used with a given model may be explored by reference to the work on SOPHIE I (Brown and Burton 1975) and the METEOROLOGY TUTOR (Brown et al. 1973) where a number of different kinds of inference are made using the same model. In the case of SOPHIE a single (numerical) circuit model is used by a number of "procedural specialists", to answer 'what if' questions, to evaluate the student's hypotheses, to list all possible hypotheses, and to evaluate requests made by the student for new measurements. These "procedural specialists" illustrate the kind of inference procedures which may be required in conjunction with specific models, and also raise a classificatory problem. Looking at specific events in the functioning of the circuit and holding a hypothesis about what is wrong with the totality of it seem to be quite distinct points of view which would be adopted in different contexts, and quite possibly for different goals. They are, however, based on the same model, being distinguished only by their inference procedures and (possible) heuristics for application. Does this mean that any difference of inference procedure always identifies a different viewpoint? This does not seem to be a particularly sensible or useful conclusion, since it would identify an explosive number of different viewpoints. It is perhaps premature to seek a clear answer to the problem at this stage, but it may be noted here since we are speaking of multiple "different" viewpoints on a domain, and may be called upon to state in what way they are different. Where viewpoints are based on different models, the distinction between them seems clear, as they are likely to have different "primitives" for description which are structured in different ways.

A similar discussion may be had in relation to the METEOROLOGY TUTOR (Brown et al. 1973) which was a precursor of SOPHIE. This system used a number of finite-state automata to produce qualitative models of meteorological state changes. The various automata are linked by "transition conditions" and also "global predicates" which represent

the assumptions behind the questions asked of it. When assembled, the automata resemble an augmented transition network. To answer specific questions, the model is set to the stated conditions and run. This produces an inference tree whose branches represent different causal chains. This may be read "root to leaf" to answer the questions detailing specific conditions, such as "If the temperature falls by three degrees ----". However, it may also be used to answer more general questions about causal relationships, such as "Does relative humidity decrease when the temperature drops?". This is done by searching for a causal path between the two named states with their relative values.

Here the different viewpoints may be characterised as quantitative and qualitative: ie. the former propagates the state changes resulting from a quantified change in the value of a specific parameter, while the latter generalises causal relationships at a different level of analysis or "grain size". Thus there are two distinct viewpoints which access the same model, but which do so in different contexts, and which use different inference mechanisms to derive their conclusions.

The kind of inference procedures that are required for our viewpoints may be obtained by defining operators of the kind used in problem solving (Newell and Simon 1972) which may be applied to the model in question to infer the desired information. As the later analysis of protocols will show, we may also wish to augment or transform the models. Other operators will be required in order to do this. These inferential operators are seen as an important part of the proper definition of a viewpoint.

The basic formulation that results from this discussion thus has three parts. The first two are an explicit domain model, and a set of operators which interrogate it. The third part encodes the knowledge about when it is useful or appropriate to apply the combined model and inference procedures. Although the work towards an implementation is initially concerned with the first two of these parts, a few more remarks about the third part are in order here so as to draw together a number points made previously. The most basic point

is that, as stated in section 2.6, our concern with the *application* of viewpoints is an issue of considerable importance for education as well as for implementation.

As Gentner and Gentner (1983) and Kieras and Bovair (1984) show, different inferences and different levels of performance may be obtained depending on the model, or lack of it, employed. The desirability or otherwise of a particular result will depend on the goals of the exercise. This indicates that different models may have different utility in relation to different goals. Young (1983) and di Sessa (1986) give examples of this when they discuss the differing utility of structural and functional models in relation to, respectively, understanding how a calculator functions, and learning to program. Effective action in pursuit of goals, or the effective application of viewpoints which have been learned, would thus seem to require that students also learn what the different combinations of model and inference procedures are good for, ie. when they should be applied.

This conclusion is supported by the discussion of Brown (1989) and Cowan (1986) in section 2.6. Brown emphasises the central role of expert practice in learning, while Cowan laments the inability of many students to apply the quantitative algorithms they had learned to problems whose nature was not immediately clear, ie. they were not good at analysing the problem to determine which algorithms should be applied. We may thus draw the conclusion that learning to apply knowledge in this way, that is learning to identify the salient features of a problem and apply a suitable model to it, is a valid educational goal, and an important component of expertise as Brown (1989) maintains. We may also express the opinion that it is a much neglected one. As noted in chapter 2, Self (1989) points to its importance when he calls for greater attention to be paid to metacognition and reflection, and recommends a separation of the task and discussion levels in relation to learning in a given domain (Cumming and Self 1989b).

An alternative attempt to formulate a structure for viewpoints is found in Wenger (1987). The language of Wenger's formulation is somewhat opaque, and we refer to it in order to

support our decision to use the language of the mental models literature in order to describe viewpoints. Wenger (1987) claims that his definition is relevant at three levels of analysis. These are in relation to specific situations, to specific domains, and as a general background. Each viewpoint has a 'kernel' composed of a number of 'keys' such as prior decisions or beliefs. Along with the 'kernel' is a 'scope' defining the "foreseeable area of relevance" of the viewpoint. In summary, Wenger sees a viewpoint as an

> "--- interpretive context whose kernel contains critical keys to the proper understanding of the entities within its scope." (Wenger op. cit. p. 355).

This "kernel" of "prior decisions" sounds remarkably like the kind of model described in Gentner and Stevens (1983). In de Kleer and Brown (1983), we might see them as the complex assumptions concerning causality and structure ever-present in naive qualitative reasoning. In Gentner and Gentner (1983), the "keys" might be the assumptions of identity between the behaviour of piped water and that of electric current. The "scope", or area of relevance may or may not be seen as intrinsic to the model. If we add an algorithm specifying just how the model is to be applied and inferences drawn in a particular situation, then we could have the bones of a "viewpoint".

Wenger fleshes out his analysis with examples of the kind of viewpoint that can be identified for each level of analysis. One would wish to argue with few of his examples. Indeed, many of them seem to echo illustrations given earlier in this paper, and he even uses the Gentner and Gentner (1983) teeming crowds/electricity analogy to illustrate one point. This then, is a generally useful analysis, but it is not clear to us why the objects analysed could not equally well be referred to as "models", thus avoiding the burden of additional terminology. Wenger (1987 chap. 15) seems to reserve this term for the totality of knowledge representations which make up the system's expertise in relation to a particular domain. Given the work in Psychology, it would seem sensible to also use the term in relation to an individual's view. "Personal" models could be distinguished from "generic" ones, while the question of "grain-size" or "level" could be dealt with by a

hierarchy of models. With respect to definitions, a range may be found in such papers as Young (1983), and de Kleer and Brown (1983). As Young points out, the structure of these may vary with the use to which the model is to be put.

Wenger does not make clear the connection between seeing the problem in a particular way, and solving it. Simply adopting a new frame of reference, or for example, a causal as opposed to a strategic ("good idea") approach does not of itself produce a solution. The new view may make a range of procedures possible. Choices must be made about which one is to be used and how it is to be applied.

3.2 The goals of the formulation.

We are now in a position to make a preliminary specification of a structure for "viewpoints" in terms of our goals for the implemented system. This structure may be considered in relation to the domain to be tutored, our design objectives, and our educational philosophy. These areas of concern are of course intimately linked, but taken separately, they may serve to structure our discussion.

For a given domain, our blueprint for viewpoints must be able to support (ie. encode) different ways of analysing that domain, so that different aspects of it are highlighted. The use of these 'modes of analysis' to support specific classes of problem-solving tasks must also be supported; (ie. the viewpoint must be able to encode the way in which different inferences may be made using the same 'mode of analysis', that is, by applying the same model). The assumption behind this requirement is that a tutoring system must itself be able to carry out the tasks or exploit the knowledge that it is trying to tutor. In addition to this, a viewpoint must be able to encode information relating to the areas where the mode of analysis and its inferencing mechanisms may be fruitfully applied.

The performance goals of the system we are designing are as follows: the implemented system should be able to conduct tutoring in a given domain in terms of two or more

viewpoints on that domain. This implies that the system should be able to tutor the viewpoints independently, and make clear their area of application, as well as tutoring their interrelationship, or use in combination. This in turn implies that the traditional problems of student modeling are made even more complicated, since diagnosis, correction, and explanation will need to be given in relation to the individual parts of a viewpoint, its use in isolation, and its use in relation to other viewpoints. While this is not the real focus of the research, it does imply that a representation of viewpoints needs to be carefully structured if the modeling and explanation tasks are to be tractable.

It has also been indicated above that the presence of multiple viewpoints may also allow us to adapt to the student (eg. adapting to their learning history or current goals) by choosing particular viewpoints for a given tutoring episode. For such adaptation to be possible, the viewpoints, or a decision process that chooses them, would have to encode information concerning the learning histories and possible goals that the viewpoints can be related to.

This desire for adaptation is scarcely separable from the educational philosophy. This philosophy (see section 2.6) demands that the student be viewed as an active partner in the educational process, so that, as stated, the tutoring is adapted to their changing goals and needs. In practice this adaptation could take various forms. If a student can be diagnosed as lacking a necessary viewpoint for a given task, then this viewpoint can be taught. If a student's goals can be either diagnosed or given as input to the system, then the viewpoints appropriate to those goals can be taught. 'Liberal' designers may allow the student to state their own needs and tutor in relation to those stated. In Brown's (1989) terms, the intention of this is that the viewpoints being taught should be 'transparent' to the student, ie. they should not be taught or learned for their own sakes alone, but should promote the student's ability to 'see through' them to the world itself.

The educational philosophy also demands that learning should be rooted in ongoing practice which is as 'realistic' as possible, and that the student should be encouraged to

consider the "discussion" or meta-level aspects of the knowledge they are dealing with. This is intended to follow some aspects of the "cognitive apprenticeship" model described by Brown, Collins, and Duguid (1989). The attention to "discussion" or "meta-level" knowledge about where a given piece of knowledge may be applied is also important for the elimination of misconceptions as Stevens, Collins and Goldin (1979) indicate. (ie. there would be little point in tutoring the 'functional' as well as the 'scriptal' views of rainfall unless the student was also induced to use them in combination). The possibility that it will be necessary to use viewpoints 'in combination' as well as individually to solve some problems, such as Minsky's (1981) car ignition example, introduces a complication: it seems that it will not be sufficient to simply list the contexts and goals in relation to which a given model and set of inference mechanisms will be useful. Ideally, the viewpoint fomulation should also enable the system to give some guidance about when and how a given viewpoint should be used in conjunction with other viewpoints.

The need to root learning in some ongoing 'practice' implies that the viewpoints, together or separately, will have to support the successful execution, as well as explanation of, relevant tasks in the domain. In other words, the system must, at some level, be able to do what it is trying to tutor.

3.3 The Formulation.

The goals expressed in sections 3.1 and 3.2 lead us to define the following working structure for viewpoints to be used as the knowledge representations of an ITS.

3.3.1 The structure for implementing viewpoints.

The viewpoints are to be composed of three modules, these are:

- The model, or set of descriptors, terms, and relationships which are used together in a particular mode of analysis in the domain.
- A set of inference mechanisms which are applied to the model in order to make the

different inferences necessary in a given mode of analysis.

• A set of representations, heuristic or otherwise, which specify contexts in which the given mode of analysis may be used, and the goals which may be satisfied by doing so.

This is represented graphically in figure 1.

Figure 1: An Outline Structure for Viewpoints.

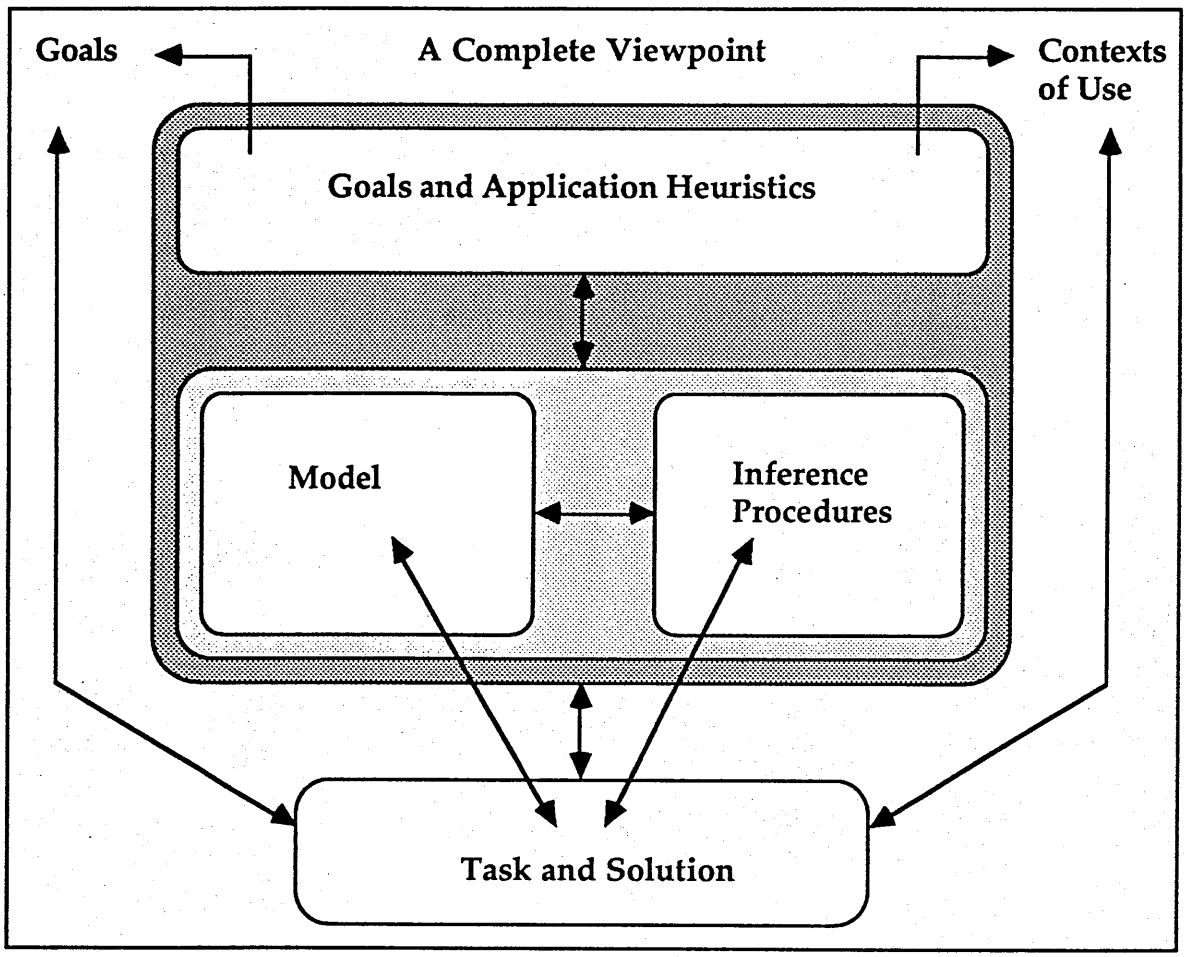


Figure 1 is intended to convey the following: A solution to a given task or problem is produced by inference procedures acting on a model of the domain in question. Specific features of the problem will map onto specific elements of the model, and onto specific parts of the inference procedures. The choice of which model and set of inference

procedures to apply to a given problem is controlled by a set of heuristics. A different formulation of these heuristics states the classes of goals (or contexts of use) which each 'model-and-set-of-inference-procedures' combination may serve. In combination, the heuristics, model, and inference procedures are referred to as a 'viewpoint'.

A simple example of this structure can be given in terms of an alternator for a car engine. First, a model of the alternator has to be described. For clarity's sake this is expressed as a set of Prolog clauses:

```
haspart(alternator, pulley).
haspart(alternator, rotor).
haspart(alternator, wires).
haspart(alternator_drive, belt).
drives(belt, pulley).
drives(pulley, rotor).
drives(rotor, current_production).
correct(belt):- undamaged(belt), tight(belt).
correct(wires):- undamaged(wires), fastened(wires).
```

This model could support a number of inferences, relating to such questions as "What parts does an alternator have?", "does an alternator have a pump?", "what drives the rotor", or, for a specific case of an alternator, "Is the belt in the correct condition?". The operators which are intended to encode the means of making such inferences form the second part of the model. These can be expressed in Prolog predicates such as:

```
drives(X, Y).
drives(X, Y):-   drives(X, Z), drives(Z, Y).
drives(X, Y):-   drives(Z, X),
                 not Y = Z,
                 message(['the processes driven by ', X, 'are outside the scope of this model']).
```

The third part of the viewpoint is a set of heuristics which guide the application of the model and associated set of inference procedures to specific problems, problem-solving

contexts, or goals. Without being coded in Prolog, these statements can be sentences of the form:

- "If the alternator is not functioning, this viewpoint can be used to identify possible mechanical causes".
- "If you wish to list the components of the alternator, use this viewpoint.".
- "If you wish to ascertain the presence or absence of a component in the alternator, use this viewpoint".

This formulation for viewpoints may be related to the goals stated in section 3.2. In terms of the domain-related goals, the 'model' will encode the different modes of analysis, or 'mental models' that are applicable. The use of this model to solve specific classes of problems is to be encoded in the inference procedures which act upon the model. This should give the resultant system the ability to actually do the tasks or exploit the knowledge that it is trying to tutor, once a suitable viewpoint has been selected for a given problem. The information necessary to make this selection is to be encoded in the heuristics which govern the applicability of models to problem-solving contexts. This is equivalent to saying that the heuristics describe the classes of goals that a model may serve. The intention that the system should be able to choose a 'suitable' model, clearly assumes that the viewpoints encoded in it are the correct and proper ones. (This is in accordance with the position stated in section 2.1.6, that the research should concentrate on the question of what could be achieved in terms of a system which utilised a number of *pre-defined* viewpoints). This does not however preclude the possibility of representing misconceptions at any level of the viewpoint structure described through such traditional methods as perturbation modeling (Carbonell 1970). The elements to be perturbed could include the model, the inference mechanisms, or the application heuristics.

The performance goals stated in 3.2 would be served through the explicit representation of two or more viewpoints in the terms described. Each may thus be tutored separately. It is also intended that the heuristics governing a viewpoint's application should encode information about its relationship to other viewpoints as well as to problem classes and

goals. The purpose of this is to give the system the ability to tutor in terms of the relationships between viewpoints, and to promote the integration desired by Stevens, Collins and Goldin (1979). How, for instance, is the mechanical view of a car alternator related to the electrical view? Does a change of state described in one view imply a change of state in the other? Is it ever necessary to use both views in combination? Which classes of problem are solvable by each viewpoint? Are there situations when it is advisable to switch from one viewpoint to the other?

The modular description in terms of model, inference procedures, and heuristics, is intended to provide a structure which will facilitate the diagnostic and student modeling functions of the system. The encoding of the heuristics in terms of both contexts where the model/inference procedure combination may be applied, and goals that may be so served, is intended to enhance the adaptability of the system. If knowledge of a student's goals can be acquired, then viewpoints related to them may be tutored. It would also be desirable that the system have the ability to diagnose when such tutoring was *not* necessary.

The explicit representation of the inference procedures as well as the models they act upon, should give the system itself the ability to function successfully in the domain it is intending to tutor. This should satisfy some of the goals implied by our educational philosophy. The student may thus be given explanation and demonstration in relation to each aspect of the domain, and may be set tasks which relate to those in the real world domain.

The explicit representation of the application heuristics and relevant goals for the viewpoint should allow the system to tutor in relation to this knowledge, since, as Cumming and Self (1989b) recommend, it separates the "discussion" or meta-level knowledge from the "task" knowledge in the models and inference procedures.

3.3.2 The viewpoint structure and "cognitive apprenticeship".

It is hoped that the structure for viewpoints given in section 3.3 will allow some learning in the style of "cognitive apprenticeship" described by Brown, Collins, and Duguid (1989). This account of learning emphasises the crucial role of "authentic" activity, and the vital contribution which may be made by aspects of the social and physical environment in which learning takes place. Four basic tutoring strategies are identified.

These are, in rough order of use:

- modeling;
- the provision of "scaffolding" for the student;
- the identification of different decompositions for a problem;
- general practice.

(These are also discussed as "modeling", "coaching", and "fading").

The idea behind "modeling" is that the tutor should demonstrate some practice in the domain as a *practitioner*, and make explicit some aspects of the tacit knowledge that underlies authentic activity. "Scaffolding" is a term used to represent the provision of a conceptual framework which the student can use to organise their thinking about the domain. This is provided at a "threadbare" level, and is given added texture by the student through practice in the domain. Such "scaffolding" may well draw on structures or systems which are already familiar to the student. The purpose of pointing out different decompositions for a problem is to indicate to the student that problem descriptions and heuristics, (and even algorithms) are not 'absolute', but may have greater or lesser utility in different contexts. The purpose of general practice is to allow the student to generalise what they have learned to new situations, so that they may become more expert practitioners, generating their own solution paths for problems, and adding more "texture" to the initially bare "scaffolding" of concepts. This "generalisation" may well also involve some re-formulation of the original "scaffolding". Such practice should be as authentic as

possible, since much of the "texture" to be added to the "scaffolding" is to be derived from interactions with the social and physical environment.

The relevance of this to the structure for viewpoints given above can best be demonstrated in terms of a specific domain, and the outline of a system intended to tutor in that domain. Let us assume that the domain is Prolog, specifically the localising of bugs in Prolog code. Let us also assume that a number of viewpoints are available which are structured as described above, and which together can describe the execution of Prolog goals and code. The purpose of the assumed system is to tutor the skill of using the viewpoints to localise bugs in pieces of code. Common sense states that the students will not be able to learn such a skill until they have some understanding of the viewpoints themselves, and can apply these to describe Prolog execution in general.

This implies three overall stages in the tutoring process for this domain:

- becoming familiar with the models and applying them to describe Prolog execution;
- becoming familiar with the viewpoints and applying them to debugging in the simpleified environment the system provides;
- generalising what has been learned to debugging in a 'real' environment.

If the assumption is made that the goals of the system are to address the first two of these stages, then the four cognitive apprenticeship strategies can be discussed in terms of how the viewpoint structure will support the strategy in relation to each stage.

The first strategy given above is modeling. For the first stage of Prolog learning, (describing execution), the system would have to model the way in which the viewpoints are applied to describe Prolog execution. This could be accomplished through a procedure which could determine which part of which model is appropriate to describe a given part of the execution. This part could then be presented to the student. The function of retrieving a specific modelpart can be allocated to an operator of the relevant viewpoint. The application of the viewpoints in the localisation of bugs is likely to be a more complex

matter, involving chains of inference. However, such 'bugfinding' could also be modeled if operators could be defined which connect a buggy behaviour to a specific piece of code through a sequence of inferences on one or more models.

The provision of "scaffolding" is the second strategy proposed by cognitive apprenticeship. In terms of the viewpoint structure outlined above, the models themselves can fill this role. For the task of describing Prolog execution, they provide a range of related and structured concepts for description, so that the problem becomes one of deciding which part of the structure to apply. For localising bugs, each model provides a means of describing and highlighting certain aspects of Prolog execution, so that the presence or absence of specific bugs can be determined. For a given buggy behaviour, this has the effect of carving up the 'problem space' of possible bugs into smaller, and hopefully more manageable, chunks. The emphasis here is that *one* of these problem spaces should provide the explanation for the bug.

The tutorial strategy of emphasising different possible problem decompositions can also be served by the proposed viewpoint structure. Depending upon the particular model and set of operators used, differing (partial) accounts of execution could be obtained, emphasising the fact that there are various ways in which execution may be described. In relation to bugfinding, the different combinations of model and operator could describe the different bugs which would all produce the same bugged behaviour. Rather than emphasising that there may be different problem spaces to consider, (as in the case of "scaffolding" above), this aspect of viewpoint use emphasises the idea that there may be several different possible explanations for a given bugged behaviour.

The strategy of giving practice so that the students become practitioners and generalise what they have learned to new situations is seen as beyond the capabilities of any tutoring system being implemented as a part of this thesis. Applying the different viewpoints to describe Prolog execution seems to be an activity which is in any case capable of little

generalisation. In the case of bugfinding, the student would hopefully generalise what they had learned by going beyond the confines and limitations of the tutoring system to apply what they had learned to localising real bugs in real code.

A few more general comments are in order here in relation to cognitive apprenticeship. The protagonists of the theory put great emphasis on the importance of "authenticity" in the activities by which the students learn. This is contrasted with the "inauthentic" nature of much schoolwork; (ie. schoolwork may well be centred on a version of knowledge which is abstracted from the activity and environment which created it, and which is studied for its own sake, rather than to achieve specific goals in the real world). This raises some question as to the extent to which viewpoints, as described above, could create an authentic environment in the virtual world of a tutoring system. Such a question can not be answered at this stage, although it is hoped that the use of formalised mental models as the core of viewpoints will bring "authenticity" a step closer.

Another question concerns the limitations of cognitive apprenticeship as a theory. Other writers, (eg. A. N. Whitehead 1932) emphasise the necessity and the benefit of occasional radical reformulations in the student's knowledge of the domain. It is not clear how cognitive apprenticeship could cater for such a reformulation, as the "scaffolding" is supposed to be clothed with detail rather that radically changed. Viewpoints, on the other hand, emphasise the alternative modes of analysis which are available for a given domain, and imply the need for such reformulations by their very existence. We will return to this issue in terms of the implementation described in later chapters.

3.3.3 The viewpoint structure and alternative possible structures.

The structure for viewpoints which is proposed above is obviously not the only one which could be adopted. The adoption of this particular structure can however be justified by considering some possible alternative structures and their potential shortcomings. Self (1990) provides a review of systems which attempt to model the holding and revision of

sets of beliefs. This review is intended to illustrate the ways in which viewpoints might be structured and implemented. The technical problems which Self shows to be involved in the construction and maintenance of such belief systems are summarised in section 2.1.6. We can also provide a critique in terms of goals that are set out in the earlier sections of this chapter. The point of this critique is to demonstrate that the use of viewpoints for *tutoring* as opposed to other activities places particular constraints on the choice of a structure to be used for implementing them.

In the systems described by Self (1990), the mechanisms used to process a viewpoint, while frequently problematic, are not seen as integral to the viewpoint. Thus the viewpoints are not 'active' in the sense of being able to produce an inference in relation to a problem. Rather, the set of beliefs is acted upon by some external mechanism. In this sense a set of beliefs is akin to a 'model', but does not give the requisite information about how that model is to be applied. As stated in section 3.2 it is desirable that the same 'mode of analysis' or model should be able to support the drawing of many different inferences via a number of different inference mechanisms, so that the tutoring system is itself able to carry out the task it is trying to tutor. Tutoring may thus be carried out either in terms of the model itself, or the inference procedures to be used in conjunction with it. Since these procedures may well differ from model to model, it seems a wise design policy to include the relevant ones with each model in the definition of a viewpoint.

Section 2.2 also states that our purpose is to design systems which can tutor in terms of two or more viewpoints on a given domain. Such tutoring can only be useful if the functionality of the distinct viewpoints can be made clear, and any misconceptions in relation to this functionality dealt with. To do this the system must have available to it some information concerning the applicability of each viewpoint. This kind of information is not generally present as an aspect of a belief system, since only appropriate problems will usually be set for it. It is true that such a set of beliefs could contain a member which states that 'the purpose of this belief set is to solve problem X', but there is no constraint

stating that such a belief *has* to be present. The purpose of 'designing-in' a section marked 'application heuristics' in each viewpoint is to ensure that this information is always present. This information will also be required if, as stated, we wish to adapt the tutoring to such factors as the student's goals. It will not be possible to choose a viewpoint to suit a given set of goals unless the system has some knowledge of what goals the different viewpoints can serve.

The goals related to the educational philosophy laid out in 3.2 also require the kind of structure which has been described in section 3.3. These state that the learning should be rooted in ongoing practice which is, as far as possible, "realistic", that tutoring should be adapted to the student's changing goals, and that they should be encouraged to consider the meta-cognitive aspects of the knowledge they are dealing with. If we are to root the learning in some ongoing 'practice', the system must be able to exploit the models it is tutoring, ie. it must also have knowledge of the relevant inference mechanisms. A 'belief system' would not necessarily have this information explicitly available. If a tutorial dialogue is to focus on the meta-level aspects of some knowledge, (eg. the context in which it is applicable), then an explicit statement of this context must be available to the system. Again, such information would not necessarily be available in a 'belief system'.

3.4 Extending and testing the proposed viewpoint structure.

Although considerable analysis and thought had gone into the development of the structure given in section 3.3, it was apparent that more work needed to be done before any effort was made to implement a system based on it. Prudence required that the real utility of the formulation be tested, while precision demanded that the inference procedures to be used as operators on the model be specified in greater detail. It was concluded that the utility of the formulation would have been demonstrated if it was able to encode the viewpoints and formalise the reasoning of real subjects in a problem-solving situation. This formalisation could then act as the basis for a performance simulation to be used in a tutoring system. It

was also concluded that a close analysis of the reasoning used in such a situation would indicate the nature of the inference mechanisms which would have to be encoded in an implemented system. This does not imply that the exercise was intended to identify specific ways of drawing inferences which could be applied to all models. Rather, the purpose was to identify *classes* of inference mechanisms which could act as elements of a viewpoint structure, and which were generaliseable. Accordingly, a study was designed to satisfy both of these objectives by recording and analysing the verbal protocols of two groups of subjects, each group being given a different model to apply to a problem domain. This study is detailed in chapter 4.

3.5 Conclusions to Chapter 3.

This chapter considered how the notion of a mental model could be augmented so as to provide a structure for representing viewpoints in an ITS. (This was not intended to imply that the resulting viewpoint should be seen as a 'psychological reality'). The need for inference mechanisms to act upon a formalised model was considered, as was Wenger's (1987) attempt to formulate a structure for viewpoints. The pedagogical importance of *applying* viewpoints was stressed. A structure was proposed for representing viewpoints in an ITS. The structure was developed by considering issues relating to the domain being tutored, the design objectives stated for the system, and the educational philosophy it is intended to embody. The structure of each viewpoint is intended to be modular, consisting of a model for the domain, a set of inference procedures to act on the model, and a set of heuristics stating the application of the model/inference procedure combination to problem contexts. This formulation was related to the 'cognitive apprenticeship' theory of learning, and to some possible alternative formulations for viewpoints. The need for a study to validate and extend the proposed formulation was stated.

Chapter 4. Testing the Formulation: A Protocol Analysis.

4.1 Introduction.

4.1.1 Goals of the study.

Chapter 3 gives an outline of a formulation that is intended to be used to implement viewpoints in an (Intelligent) Tutoring System. The structures of the formulation were developed by considering issues relating to the domain being tutored, the design objectives stated for the system, and the educational philosophy that the system is intended to embody. The structure of each viewpoint is intended to be modular, consisting of a model of the domain, a set of inference procedures to act on the model, and a set of heuristics stating the application of the model/inference procedure combination to problem contexts. The chapter concludes with a statement of the need for a study to validate and extend the proposed formulation. This study is described below.

This study was motivated by two related goals. Firstly, we wished to establish a conceptual basis for implementing viewpoints in Intelligent Tutoring Systems. This involved demonstrating that two different viewpoints and the reasoning associated with them could be formalised in terms of a single structure of context, models, and operators. In terms of the study described below, we wished to show that our single structure for viewpoints would allow us to formalise the different reasoning patterns that are observed when two different viewpoints are applied to the same system. Success in this endeavour would be taken as confirmation that the proposed structure was able to formalise a range of viewpoints and the reasoning associated with them. This formalisation method could then be used as the basis for implementing viewpoints in a tutoring system.

It was thus necessary to identify the observable differences in reasoning patterns in more specific detail than previous work had done, in order to formalise them. The study

described below uses protocol analysis to distinguish the different reasoning patterns associated with two quite distinct mental models. In chapter 5 we then use the single proposed structure for viewpoints to formalise these different patterns. It is during this exercise that the second goal of the study is pursued, that of obtaining a more precise definition of the kinds of operator which would be required for the actual implementation of a viewpoint.

4.1.2 Outline of the study method.

Ten pairs of subjects were given the task of collaboratively operating a computer simulation. Half the pairs were given a functional model of the system, and half a structural model. Their discussions were recorded and analysed. The different inferential processes the analysis revealed were then formalised using the framework of model and operators outlined above. The requisite operators were developed after the protocols had been analysed, and can be shown to fall into three distinct classes.

'Structural' and 'Functional' models were chosen since they are two of the most commonly distinguished types of mental model. (eg. di Sessa 1986). They are described in detail by Young (1983) as "Surrogate" and "Task/Action Mapping" models. A structural model describes the structure of the device in question, while a functional model describes the use of the device in achieving specific operational goals.

The method of applying two different models to a single device draws on the work of Gentner and Gentner (1983), and Kieras and Bovair (1984). The domain of a 'power generation system' was chosen because firstly, it lent itself to description in terms of the two models, and secondly, it was judged to have the right level of familiarity for the intended subjects. It was sufficiently familiar that they could reason about it in the terms they were given with some feeling of confidence, but it was not so familiar that they would

already have well-established and coherent models of this kind of system. (This was based on the assumption that the intended subjects who were students, technical, and academic staff at the Institute of Educational Technology would not have a working knowledge of power generation systems).

4.2 Details of the study method.

4.2.1 Models and Instructions.

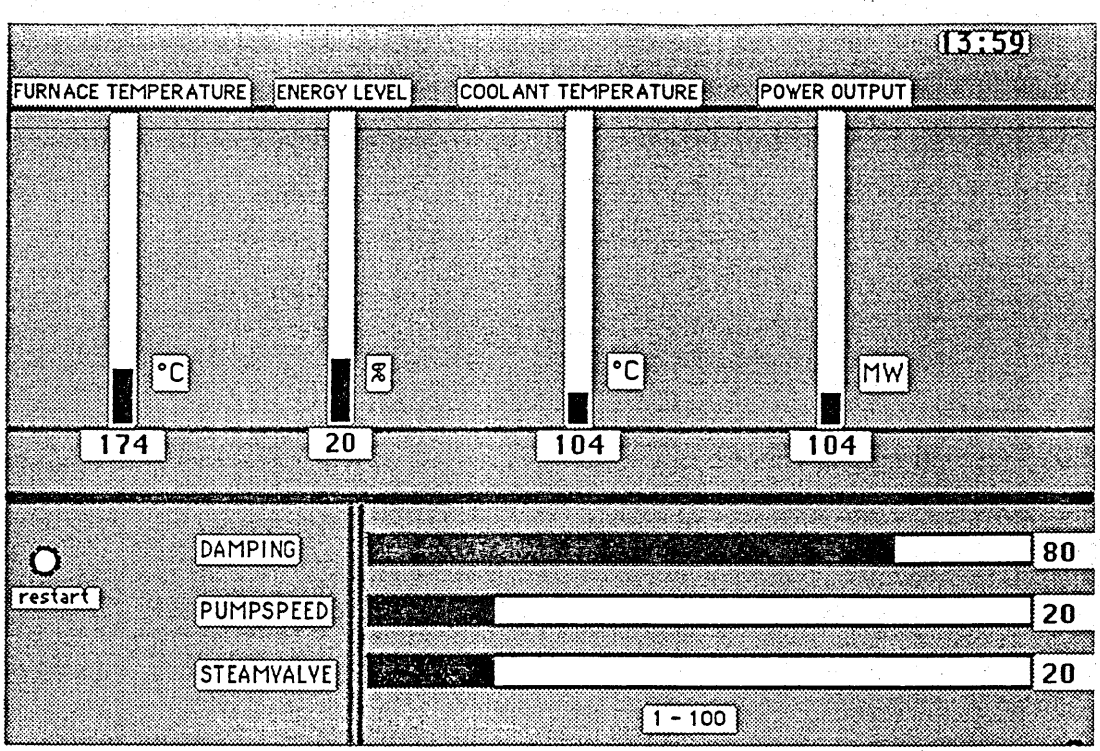
The control panel for a much-simplified (nuclear) power station was built in Hypercard on an Apple Macintosh; (see figure 2). Ten pairs of subjects, (five pairs for each model), were given either a Functional or a Structural model (di Sessa 1986) to apply to the simulation, each member of a pair being given the same model. Jointly-responsible pairs were used on the assumption that they would justify their actions to each other, and thus articulate their reasoning.

We stress here that we are not using 'functional' in the sense employed by de Kleer, (eg. de Kleer and Brown 1983). These authors view any knowledge of 'function' as being crucially connected to an appreciation of structure in the relevant system via notions of causality. The whole point about a 'functional' model as described by Young (1983), is that it does not entail any notion of causal mechanism, since it has no description of the internal structure of the device in question. The work by de Kleer, while fundamental, is of only limited relevance here. In their (1983) paper, de Kleer and Brown discuss the possibility of having different embedded models of the same device, and state that:

> " Unfortunately, our theory has no mechanism to handle or to profit from this situation; nor does it say anything about multiple device topologies." de Kleer and Brown (1983) p. 188.

It is just these multiple models and multiple topologies that the current work on viewpoints is intended to explore. de Kleer and Brown take the basic components of the system to be described as given. The thrust of our work is to investigate the different sets of 'primitives', or 'conceptual components' which may be used to describe the same system. While de Kleer and Brown have the tenet of "no function in structure" as a central part of their theory, the whole purpose of our work is to investigate how a given system may be conceptualised in different ways in order to pursue different functions.

Figure 2: The simulation screen in starting state.



In the study reported here, the Functional model described the device as the control panel for a power generation plant, and gave a list of control movements which would achieve operational goals, such as raising the "furnace temperature". The Structural model described the device as the control panel for a nuclear power station, and briefly described the flow of energy through the system.

All subjects were presented with a set of written instructions, conveying the model they were being asked to use, and details of the goals they were to achieve using the computer simulation.

The instructions based on a Functional model read thus:

"Please imagine that you and the other person in the study are jointly responsible for the safe operation of the industrial plant described below. You both see the same screen displayed, but only one of you can operate the controls. You can communicate by talking normally. If you have a mouse to operate the controls, please think aloud, and explain to the other person what you want to do, and why, before you do it. If you cannot operate the controls, please make your thinking clear to the other operator. Please respond to any instructions which appear on the screen.

The horizontal bars at the bottom of the screen are the controls, while the vertical bars at the top give information about the state of the system. Please study the printed illustration and say when you are ready to continue.

The plant MUST be operated within the constraints stated below.

- -

This device is the control panel for a power generation plant. The controls are used to carry out the following functions by executing the actions given on the right.

To increase furnace temperature: ->	decrease damping or decrease pump speed or both.
To reduce furnace temperature: ->	increase damping or increase pump speed or both.
To increase power output: ->	open steam valve.
To decrease power output: ->	close steam valve.

Furnace Temperature must be between: 500 - 600 °C.

Power Output must be between: 600 - 800 MW."

The instructions based on a Structural model read:

"Please imagine that you and the other person in the study are jointly responsible for the safe operation of the industrial plant described below. You both see the same screen displayed, but only one of you can operate the controls. You can communicate by talking normally. If you have a mouse to operate the controls, please think aloud, and explain to the other person what you want to do, and why, before you do it. If you cannot operate the controls, please make your thinking clear to the other operator. Please respond to any instructions which appear on the screen.

The horizontal bars at the bottom of the screen are the controls, while the vertical bars at the top give information about the state of the system. Please study the printed illustration and say when you are ready to continue.

The plant MUST be operated within the constraints stated below.

- -

This device is the control panel for a nuclear power station. It is a fission reactor, with the reaction controlled by damping material. The energy is drawn from the reactor core by means of a pumped coolant. The hot coolant is used to produce high-pressure steam which is fed to a turbine through a control valve. The turbine turns a generator, which produces electrical current.

Furnace Temperature must be between: 500 - 600 °C.

Power Output must be between: 600 - 800 MW."

It may be noted that in the 'structural' model, no information about how to achieve specific operational goals using the controls was given. Formalised versions of these models are given in section 5.5.1.

Apart from the differing models all subjects were given the same instructions, along with a labeled printout of the "control panel" they were about to use. (See figure 2).

As the instructions indicate, the members of each pair were asked to be jointly responsible for operating the system within specific constraints, (500 - 600 °C. for furnace temperature, and 600 - 800 MW. for power output), and told that these constraints MUST be adhered to. They were seated at back-to-back monitors with a partition between them, so that they could communicate verbally, but not visually. Each monitor showed the same information, but only one member of each pair had a mouse with which to control the system. The mouse-owners were instructed to negotiate their control actions with the mouseless partner, and to describe each action and its rationale while carrying it out. The mouseless partners were instructed to participate in this negotiation, and influence the mouseowners as they saw fit. This arrangement was intended to stimulate the production of think-aloud protocols. These think-aloud protocols were recorded, transcribed verbatim, and analysed.

4.2.2 Training Systems.

Before the main session, all pairs were given the same training on two simpler systems, which comprised one control bar and one readout bar, in the same orientation to the screen as the main study's bars; (controls horizontal, readouts vertical. See figure 2). These were said to represent a compressed-air tank which could be set to various pressures using the control bar. The first training system established mouse skills and inter-subject familiarity, and was stable in the sense that once the system had reached the level set in the control bar, it would stay there. The control and readout bars were both on a scale from one to a hundred, and the pressure value set was the one which finally appeared in the readout.

In the second training system, the compressed-air tank had to be kept within certain pressures, and was said to be part of a larger industrial process. It would move by small

steps to the pressure set in the bar, and stay there for a short time. If after some 25 seconds, no new pressure had been set, the displayed pressure would start to rise quite quickly, necessitating remedial action on the part of the operator. This prepared the subjects for instability in the main system. Also, while the control scale remained 1 - 100, the readout was on a scale from 1 - 1000. The pressure value displayed was calculated by an equation which made it proportional to the internal control value.

4.2.3 The Simulation Screen.

The screen for the main session showed three long horizontal bars, and four shorter vertical ones above them. (See figure 2). The horizontal bars acted as controls, displaying the control settings, and accepting new control settings via mouseclicks in the bar. The vertical bars gave a readout of various parameters for the system. The three controls were for damping, pumpspeed, and steamvalve setting. The readouts were for furnace temperature, energy level, coolant temperature, and power output. All bars were labeled according to their function, and had a numerical, as well as a graphical, readout. These labels were chosen so as to be neutral with respect to the type of energy source powering the system. All sessions were started with the simulation in the same state, with the furnace temperature and power output readings well below the levels set by the constraints.

4.2.4 The Simulation Algorithms.

The various readouts, (Furnace Temperature, Energy level, Coolant temperature, and Power Output), were calculated from the control settings. It was intended that the algorithms should maintain the basic logic of heat and energy flows in the type of system described. Thus if the Coolant Temperature dropped below 100, the Power Output immediately went to zero. (ie. there was no steam available to generate power). If the

Steamvalve was opened, the Power Output would increase, and the Coolant Temperature would drop slightly unless other parts of the system were adjusted.

To simulate the inertia likely to be found in real industrial plant, the system had a fairly ponderous and measured response to changes in the control settings. It could take some time for them to be fully reflected in the readout values. The "steamvalve" gave the quickest response, being quickly reflected in changing "power output" values.

If a pair manouvered their system into a stable state, it would eventually de-stabilise itself so that the session would continue to yield information about the pair's inferences. After the system had been stable, (ie. without a change of more than 10% in any parameter), for some 40 seconds, the internal value of either damping or pumpspeed was incrementally changed, so that the system started to heat up. The corresponding readout values were calculated and displayed, although no change was made to the control settings shown on the horizontal bars. This destabilising loop would retain control until either the mouse was clicked in the relevant control, or the maximum internal value for that control was reached. This de-stabilisation applied at all times.

At the end of the twenty minutes set as the time for the main session, the system took control of the display, refusing any new control settings, and raising all readout values to a high level. The following message then appeared on the screen:

"Please state what you think this power generation system will do next, and why."

The question remained on the screen, and the system took no further action. The subjects were allowed free discussion at this point, which was recorded until the subjects appeared to have nothing further to say.

4.2.5 The Subjects.

The subjects were research students, secretaries, technicians and staff at the Open University campus in Milton Keynes. Twenty people took part, (eleven men and nine women), randomly allocated to ten pairs, (five in each condition). At the start of each session, a coin was tossed to see which of the pair would control the mouse.

4.2.6 Expected results and purpose of the study.

This is based on the properties of the models referred to in the previous section. Logic dictates that if some subjects are informed about the structure of a system, but not about how to achieve specific operational goals with it, then they will have to make some assumptions and produce a chain of reasoning which links their current model to a specific course of action; (ie. they will have to develop their own functional model). Knowing, for instance, that the system is controlled by damping material and has a pumped coolant does not immediately tell the subjects how to keep the furnace temperature within the constraints set. We may predict this reasoning process without making any strong commitment as to the particular psychological mechanism involved. The group using a functional model do not need to carry out this reasoning, since their model already links specific actions to the explicit operational goals of the simulation. We would thus not expect to find evidence of this reasoning in their protocols.

We wish to show that, (for the purposes of ITS design), the two distinct viewpoints and reasoning exhibited by the two groups can be formalised using the *single structure* for viewpoints described above; ie. by a formalisation of the models they were given, with appropriate operators. The success of this exercise should demonstrate the utility of our formulation of viewpoints as a starting point for implementing them in our systems; ie. we

can use the same viewpoint structure to formalise the various viewpoints that are required for the tutoring system.

4.3 The Protocol Analysis.

4.3.1 The differential analysis.

The purpose of the analysis was to demonstrate that there was indeed a difference between the reasoning of the two groups, as Gentner and Gentner (1983) would predict. The recorded sessions were transcribed as verbatim protocols. We wished to find a single encoding category which would capture the difference of reasoning style referred to above; (ie. there should be a significant difference between the two groups, (or at least an indicative trend) in the number of protocol segments encoded in the category. The group using a structural model were not told explicitly how the various parts of the system interrelated. They had to make assumptions based on their knowledge of the world and reason causally from these. These assumptions appeared to involve *objects* such as "air" in the furnace, or *processes* such as "fission reaction" or "meltdown". The point of interest was that these *objects* and *processes* were not represented on the monitor screen or in the instructions, (see fig. 2 for an example of the simulation screen), and had to be imported by the individuals concerned along with assumptions about their relationships. To capture these importations and assumptions the encoding category is defined as follows:

"Descriptions of the power generation system using knowledge of real-world objects or processes which are not represented on the screen."

The fact that an object such as "coolant" or "steam" was mentioned in the structural model should not automatically exclude a phrase mentioning it from being counted as an example of the encoding category, since the information given in the instructions is never sufficient to support causal reasoning directly: eg. the existence of a "pumped coolant" does not tell

the subject how it interacts with the other parts of the system, so that assumptions about this still have to be made.

The following protocol segments are examples of the encoding category:

(1) "Yes, increase the air in the furnace".

(2) "I have increased the speed of the coolant going round the system and it's bringing down the temperature again".

The first example fits the category as the screen has no representation of air in the furnace. The second example fits because while the screen does give information about two different temperatures, it has no representation of coolant circulating around some system.

The assumptions made here can be described roughly as:

1) the coolant circulates round the system;

2) the speed of this circulation is directly related to the setting of the "pumpspeed" control;

3) making the coolant circulate faster takes more heat from the furnace and so brings its temperature down.

Statements concerning the stability of the system, and generalised statements to the effect that "something is going to happen" or "might happen" were not encoded as instances of the category.

4.3.2 Segmentation and Validation.

The transcripts were divided into segments which were then encoded by the experimenter as either being, or not being, an instance of the category described. The reliability of this

encoding was checked by having three independent judges each encode four of the transcripts using the category, examples, and exclusions stated above.

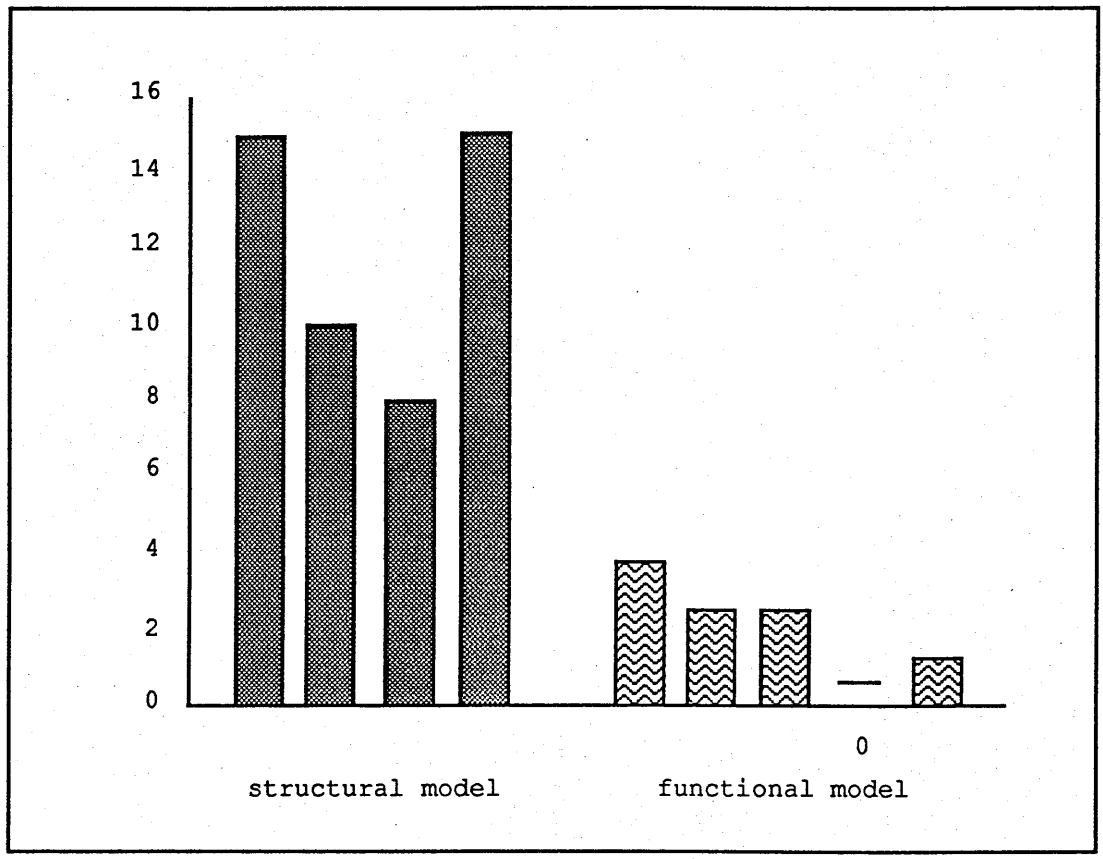
The judges were all given the same transcripts, two from each subject group, with no indication as to what kind of group the transcripts came from. The resulting encodings were not discussed with the judges. In order that the transcripts and the encoding category should be meaningful, the judges were introduced to the simulation and its two models. The results of this showed a clear agreement with the investigator on at least 75% of the encoded segments, without discussion. This was taken as a confirmation that the investigator's own encoding was sufficiently reliable.

Where possible, the segment boundaries were made identical with clause or sentence boundaries. Where sentence structure was incomplete, natural breaks in the dialogue such as the change from one speaker to another were used. Where two clauses were linked by conjunctions such as "so" or "while", and the experimenter judged that there was a material difference of content between the clauses, a segment boundary was inserted on one or other side of the conjunction.

4.3.3 Protocol Analysis Results.

Nine protocols were available for analysis, four from the structural group and five from the functional group. The tenth protocol was unavailable due to a technical fault in the recording apparatus. The number of encodings for the specified category in the first twenty minutes of each session is shown in figure 3. The number of subjects is too small for meaningful significance (t) testing, but the trend for a greater number of encodings among the group using a structural model appears to be strong.

Figure 3. The number of encodings of the specified category in the first twenty minutes of each session for each pair of subjects. Grey columns represent the pairs given a structural model. Patterned columns represent the pairs given a functional model.



4.4 Discussion of the Protocol Analysis Results.

4.4.1 Introduction.

Figure 3. indicates that the subjects using a structural model interpret the system in terms of their imported knowledge of real-world objects and processes to a much greater extent than do the subjects given a functional model. We will now attempt to show that the main use of this imported knowledge is to support causal reasoning, and that this involves a different reasoning procedure to that employed with the functional model. Having established the

distinction, we will indicate how our proposed structure for viewpoints may be used to formalise examples of the reasoning of the two groups.

We structure this analysis by looking at the way the two groups reasoned about the controls they could manipulate, the steamvalve, the damping control, and the pumpspeed control. (See Figure 2 for an example of the simulation screen).

4.4.2 The Steamvalve Control.

We start by considering how the two groups reason about changing the power output levels of the system. The following excerpt shows a pair with a structural model discussing their lack of power output:

> S1. "The energy level is up, so we should be able to get power output shouldn't we?"
> S2. "No, because you can't, because you don't have the steam. The coolant is not hot. You can't get any steam to produce the power."

The subjects' model only tells them that "the hot coolant is used to produce high-pressure steam which is fed to a turbine through a control valve". It requires reference to the real-world knowledge that steam can only be produced above the boiling point of the water, for the subjects to develop a functional model of how to raise power output. They have to make the assumption that it is possible for the coolant temperature to be too low to produce steam for the turbine. When, by dint of luck or reasoning, the subjects have raised the coolant temperature, it is necessary for them to assume that the "control valve" of the model is in fact the bar labelled "steam valve" on the screen, and that increasing the numerical value of the setting for this is analogous to opening the valve. Another piece of real-world knowledge tells them that opening a valve allows more of the controlled material to pass through. These last inferences involving the valve can be seen a few lines after the previous quotation, where S1. says:

"The energy level is really high now, why aren't we getting any of it as power output? The steam valve. The coolant temperature's right up now, lets have more steam valve".

These subjects thus seem to build a functional model, on top of the structural one they were given, by a process of causal reasoning.

This process may be summarised as follows:

1) Steam is generated at the boiling point of the water.

2) The source of heat for this process is the system coolant.

3) Therefore the coolant must be hot to cause the generation of steam.

4) This steam will only cause power to be generated if it is fed to a turbine.

5) The control labeled "steam valve" controls this flow of steam.

6) Raising the setting of the steam valve will cause more steam to flow through it and thus raise power output by causing the turbine to spin.

In terms of our suggested structure for viewpoints, the assumptions about how to raise power output, described above, could be modeled by operators which augment the structural model with the relevant piece of knowledge, and attempt to draw inferences from the result. This is exemplified in more detail below in relation to a different quotation.

The process for the group who are given a functional model is less complex, and does not involve causal reasoning or reference to real-world knowledge which is not provided in their model. Examples of this are,

S1. "Power is going down, we need to increase the steam valve".

S2. "So increase it now, Oh".

S1. "Too much".

and,

S1. "Oh dear, we are getting nothing like the power we need. Open steam valve up about, put it up to fifty".

These may be characterised as "condition-action" pairs, echoing the content of the functional model the subjects were given. This reasoning is not characterised by the reference to causality and physical processes found in the structural model groups.

A direct manifestation of the functional model they were given occurs when the dialogue takes the form of goal-action pairs such as:

S2. "It's going too quick, isn't it.
S1. "What on the power? so reduce power output, I'll decrease steam valve slightly here".

The form of reasoning here may by characterised as far more rule-based than causal-based, so that the subjects simply have to find the goal in the model which corresponds with their own goal, and select one of the actions given for that goal.

A variant of this for the condition-action pairs involves selecting a goal which will change the unwanted condition to a desired one, (usually the inverse of the unwanted condition), finding the goal in the model which corresponds with that goal, and selecting one of the actions given for that goal.

4.4.3 The Damping Control.

A similar pattern is found here. The structural group reasoned about the damping control through various forms of real-world knowledge. One subject declared their intention as follows:

S1. "I'm going to try and stop the furnace from cooling too much, by clicking on the damping, allowing it a bit more air by going down towards the zero".

It would appear that the "furnace" label led this subject to elaborate their model of the system with knowledge of a conventional furnace, and reason in these terms, in spite of being told that this was a nuclear power station. Was the subject reasoning in terms of the domestic coal fire, whose rate of combustion is regulated by controlling the flow of air to it?

Other subjects did not instantiate the model's "damping material" as a particular substance, but used their concept of damping as a process which can have a greater or lesser effect to make assumptions and deduce a functional model for this control. The following three quotations illustrate this:

> 1) "It's jumped up again, it wants more damping. Slow down the fission reaction".
>
> 2) "Take off the damping slightly, then you create more reaction. Hopefully that will increase the cooling".
>
> 3) "Reduce the damping 'cause it must be damping the reaction."

It is interesting to note in the second quotation how the application of a piece of knowledge has produced a deduction in direct opposition to the actual effect of the change, (ie. the hope that more reaction will increase the cooling). If the functional-model groups showed this reversal at all, it was quickly corrected, and generally associated with clear statements that they were confused.

As with the steamvalve, the group with the functional model generally justified their decisions in instrumental rather than causal terms, reasoning within the scope of the model they had been given, as in

S1. "Shall we reduce damping?"
S2. "To increase the furnace temperature, yes, and maybe a lot."

4.4.4 The Pumpspeed Control.

A similar pattern is found in relation to the pumpspeed. Faced with a falling coolant temperature and power output, one structural model pair respond thus:

S1. "Coolant temperature is going down rapidly, so decrease pumpspeed."
S2. "To there?"
S1. "Yes, ... That means it stays in for longer and gets warm."

What in the first two lines seems to be a functional exchange has a clear causal justification attached to it in the third. The subject's model only tells them that the system has a "pumped coolant". A passing knowledge of the cooling system of a water-cooled car engine would be sufficient to make the necessary assumptions and reason about the effects of changes in the pump speed. (It is of course also necessary to assume that "pumpspeed" control affects the "pumped coolant". Subjects appear to do this automatically.)

A different "structural" subject, faced with a similar situation, appears to use the same expanded model, but with less certainty:

"When it was going really hot, it did make it to the power output, so try giving the pumpspeed a big whack upwards. I can't figure out whether rushing more coolant past improves it or makes things worse."

This last speaker explicitly states in the discussion at the end of the session that he used his knowledge of a car engine cooling system to make inferences about the reactor.

4.4.5 Errors in the reasoning.

The structural model group came to mistaken conclusions about the causality of the system far more frequently than the functional model group. This has been referred to above, and it is evident in the last quotation, although this specific aspect of the protocol analysis has not been verified. It does seem to be a reasonable result if we accept that the structural group were reasoning about the system more frequently in causal terms, and were looking for importable knowledge which might be used as assumptions to illuminate their situation. Another example is:

> "... It should go up now. Maybe the pumpspeed is very dependent on how much steam is coming through."

This is mechanically possible, and in that sense plausible. However, if it assumes that the pumps are using the very energy the system is generating, this would seem like a rather savage feedback loop which might have the system oscillating between very hot and very cold. Alternatively, the speaker might simply have failed to consider this matter. The point is, that this group of subjects are evidently trying to augment their model with causal links so as to deduce the correct functional model for the constraints they have been given. With sufficient time, we would expect them to do this quite successfully.

4.4.6 Responses to extreme system states.

A different kind of "error" was produced in response to extreme system state readings. The instability routines sometimes produced extremely high readings for the furnace temperature, energy level, and power output controls. These produced different reactions in the two groups, and although this part of the protocol analysis has not been verified, the results are interesting.

In these post-Chernobyl days, ideas such as "meltdown" and "going critical" may be taken as common knowledge. In the first twenty minutes of the session, reference to these ideas are far more common among the pairs with a structural model. Examples are:

1) "I don't want it to go critical at this stage."

2) "Meltdown."

3) "We have just irradiated the world".

These remarks were usually made in response to the furnace temperature going well above its upper constraint. The structural group were told that they were dealing with a nuclear reactor, and appear to make the following assumption: This reactor is going to behave in the same way as reactors in the real world. They import the knowledge or belief that when reactors get particularly hot they get particularly dangerous, and make the appropriate inference. The operating constraints they are given, and the bar-chart scale appear to provide the context for judging what is or is not "particularly hot".

The group with a functional model respond rather differently to exceptionally high readouts. They are generally more concerned with high power output readings than with high furnace temperatures. Typical statements are:

1) "We are going to self-destruct here"

2) "We've just blown apart"

3) "But it's going to explode if the power is too high".

These show the importation of a different kind of world knowledge and provide some of the few examples of the encoding category found in the functional group transcripts. They

were told that they were dealing with a "power generation system", and appear to make the following assumption: This is a mechanical system which is going to behave like mechanical Systems in the real world. They import the belief that when systems are stressed far beyond their operating levels, they may self-destruct or explode, and make the appropriate deduction. Thoughts of "irradiation" are absent. There is no particular reason to focus on the furnace as the most dangerous component. Their sensitivity to power output levels can be understood if we remember that these levels are controlled by manipulating a steam valve. If the assumption is made that excessive power output levels must mean excessive quantities of steam, and this is coupled with the imported knowledge that steam is usually harnessed by having it under great pressure, their fears appear quite rational.

4.4.7 Problems involving the simulated system.

Some aspects of the hypercard simulation caused confusions which may have lead to less distinct results than might otherwise have been obtained. One of these was the overall speed of the Macintosh Plus running the program. The simulation was implemented as a sizeable piece of code, (a Hypertalk script) which caused the machine to run rather slowly so that sometimes it could not keep pace with the rate of clicks being made in the control bars. This led to two kinds of confusion in the subjects. Firstly, the subjects wondered if the machine was reacting to them at all, and would automatically click twice or three times. This only exacerbated the problem, since when the program did get around to taking input from the clicks, the cursor could well be in another part of the control bar or screen, and it was to this later position that the program responded. In this situation the experimenter told the subjects to click only once, and keep the cursor in the same position until the machine had responded to the input.

Other problems were caused by the programmed instability of the simulation. In retrospect, it would have been wiser to have this operating only when the system was within its stated constraints, rather than at all times. This would have continued to provide information about the subjects inferences without causing unnecessary confusion. As it was, the subjects sometimes felt that the behaviour of the machine contradicted their expectations, and thus inhibited their reasoning about how to control it. The fact that there were two kinds of instability, and that the particular one invoked would continue until an adjustment was made to the control which was related to it, only complicated matters.

4.5 Conclusions to Chapter 4.

As noted previously, the goals of this study were to identify different reasoning patterns associated with the use of different models, so that these differences could subsequently be formalised using the proposed conceptual structure for implementing viewpoints in an ITS; (see chapter 3). This structure consists of a model, a set of operators which draw inferences from it, and a set of heuristics which indicate the viewpoint's area of application. Success in this exercise would be taken as confirmation that the proposed viewpoint structure was able to formalise a range of viewpoints for implementation in an ITS.

A protocol analysis study observed two groups of subjects applying either a functional or a structural model to a simulated power station. Their verbal protocols were recorded and analysed. This analysis identified a trend towards causal thinking based on imported real-world knowledge which is associated with the use of a structural model in the situation studied. The subjects appeared to use to use the imported knowledge to build personal, composite models which are not necessarily coherent. The functional model appears to involve the use of condition-action or goal-action based reasoning with much less reference to imported knowledge.

Stronger conclusions concerning the different inference processes involved in the use of different models may not be drawn, due to the limited numbers in each group of subjects. Also, no claims are made as to the precise psychological mechanism, (eg. analogy, qualitative reasoning), involved in the manipulation of the models described. The question of heuristics to select a given viewpoint as appropriate to a specific problem has not been investigated here.

The fact that the group given a structural model tended to reason about the simulated system causally, while those given a functional model tended to reason in a rule-based, condition-action way supports the proposal that different models do have different utility, as Minsky (1981) and the designers of STEAMER (Hollan et al. 1984) suggest. We may imagine two groups of people, one whose goal was to repair the power system when it was faulty, and another whose goal was to operate it safely under normal conditions. We would suggest that the structural, (causal), view was appropriate to the repairers, as it would allow them to reason about the system's behaviour under abnormal conditions and make predictions about its behaviour. The 'operators' would not necessarily need any such causal perspective, since they could, (as in the study), simply follow the rules for different conditions. How many individuals after all, happily operate a motor car or a computer without any detailed knowledge of its mechanism? The point of these remarks is that they have some implications for tutoring, since if a student's goal can be identified, then a viewpoint appropriate to it may be selected, (given that one is available). Put another way, this may help the system to select problems and exercises which are meaningful to the student.

Chapter 5. Testing the Formulation: Formalising the results of the Protocol Analysis.

5.1 Introduction: The Goals of the Formalisation.

The study described in chapter 4 was motivated by two related goals. Firstly we wished to establish a conceptual basis for implementing viewpoints in Intelligent Tutoring Systems. This involved demonstrating that two different viewpoints and the reasoning associated with them could be formalised in terms of a single structure of context, models, and operators. In terms of the study described in chapter 4, we wished to show that the single structure for viewpoints described in chapter 3 would allow us to formalise the different reasoning patterns that are observed when two different viewpoints are applied to the same simulated system. Success in this endeavour would be taken as confirmation that the proposed structure was able to formalise a range of viewpoints and the reasoning associated with them. This formalisation method could then be used as the basis for implementing viewpoints in a tutoring system.

It was thus necessary to identify the observable differences in reasoning patterns in more specific detail than previous work had done, in order to formalise them. The study described in chapter 4 uses protocol analysis to distinguish the different reasoning patterns associated with two quite distinct mental models. The work described in this chapter uses the single proposed structure for viewpoints to formalise these different patterns.

The second goal of the study was to arrive at a more precise definition of the kinds of operator which would be required for the actual implementation of a viewpoint. This goal was pursued by considering the detail of the formalisations of the different reasoning patterns.

As stated in chapter 1, the inferential operators that interrogate a model are seen as vital to the proper definition of a viewpoint for implementation. One reason for this lies in the

belief that in order to tutor and explain a domain effectively a tutoring system must itself be able to perform the tutored tasks in the given domain. It must thus be able to draw the required inferences from a given model and so must be equipped with suitable means of doing so. Since the structure for viewpoints was developed with the intention that it should be used as a basis for formalising viewpoints which could then be implemented as the domain representations of an ITS, it was thought wise, before starting the implementation, to actually test the structure's ability to formalise different viewpoints on a domain, and develop it as necessary.

This resulted in the definition of three classes of operator which act on a given model. The first of these simply retrieves explicit elements of the model. The second draws inferences which are implicit in the model by applying an inference procedure to two explicit elements of the model. The third augments (or transforms) the model with new information or assumptions. The first two classes of operator may then be applied to the result.

In summary, the formalisation described in this chapter had two purposes:

- to determine whether the intended structure for viewpoints was adequate. If the models used by different groups and the inference procedures associated with each model could be shown to be quite distinct, and if the intended structure for viewpoints could be used to formalise *both* combinations of model-and-set-of-inference-procedures, then it could be concluded that the intended structure was sufficiently robust to serve as the basis of an implementation.
- to determine what kinds of operator would be required. The notion of an 'operator' is a very general one. The implementation of a system requires that specific operations be defined. In order to fulfil our goals for the tutoring system design we needed these operations to reflect, as far as possible, the reasoning used by human users. It was assumed that the analysis and formalisation of reasoning exhibited by human subjects could help to define classes of operator which satisfied this need.

The issue of how to formalise the heuristics which describe the area of application for each viewpoint is not dealt with in detail in this chapter, as the formalisation of the operators and reasoning processes was seen as a prior task.

5.2 Formalising the reasoning of the subjects.

5.2.1 A Formalisation of the models.

Having established the distinction between the reasoning patterns of the two groups in chapter 4, we will now indicate how the our proposed structure for viewpoints may be used to formalise this reasoning. As stated earlier, this formalisation makes no claims to be psychologically valid. Each model will be expressed as a sequence of Prolog clauses. (All Prolog examples here and elsewhere in this thesis are based on the syntax and predicates of LPA MacPROLOG version 3.0).

The functional model formalised in Prolog.

```
exists( power_generation_plant ).
haspart( power_generation_plant, control ).
iscontrol( damping ).
iscontrol( pumpspeed ).
iscontrol( steamvalve ).
increase( furnace_temperature ):- decrease( damping ).
increase( furnace_temperature ):- decrease( pumpspeed ).
increase( furnace_temperature ):- decrease(damping), decrease(pumpspeed).
decrease( furnace_temperature ):- increase( damping ).
decrease( furnace_temperature ):- increase( pumpspeed ).
decrease( furnace_temperature ):- increase(damping), increase(pumpspeed).
increase( poweroutput ):- increase( steamvalve ).
```

```
decrease( poweroutput ):-  decrease( steamvalve ).
legal_poweroutput( X ):-  X > 600, X < 800.
legal_furnacetemp( X ):-  X > 500, X < 600.
inconstraints( X, Y):-  legal_poweroutput( X ), legal_furnacetemp( Y ).
```

The Structural Model Formalised in Prolog.

```
exists( nuclear_power_station ).
haspart( nuclear_power_station, reactor ).
haspart( reactor, damping_material ).
haspart( reactor, coolant ).
haspart( reactor, pumps ).
haspart( nuclear_power_station, control_valve ).
haspart( nuclear_power_station, turbine ).
haspart( nuclear_power_station, generator ).
controls( damping_material, reactor ).
moves( pumps, coolant ).
takes_energy( coolant, reactor ).
makes_steam( coolant):- hot( coolant ).
controls( control_valve, steam ).
turns( steam, turbine ).
turns( turbine, generator ).
produces( generator, electricity ).
legal_poweroutput( X ):-  X > 600, X < 800.
legal_furnacetemp( X ):-  X > 500, X < 600.
inconstraints( X, Y ):-  legal_poweroutput( X ), legal_furnacetemp( Y ).
```

It is evident from the Prolog clauses that these models contain quite distinct sets of information, and that each can answer questions unanswerable by the other. Eg. the

question "How do you increase the furnace temperature?" is directly answerable by the formalised functional model, but not by the structural model, since the latter has no representation of "increase". The structural model can, however, answer the question "What moves the coolant?", while the functional model cannot. The functional model has no representation of pumps, coolant, or their relationship. This indicates that the two models have different utility in relation to different sets of goals, such as predicting system behaviour, or controlling specific parameters.

5.2.2 General remarks concerning the formalisation.

We now demonstrate how the reasoning shown by our subjects can be formalised using our proposed structure for viewpoints. The demands of the formalisation help to clarify the classes of operators that are required by the viewpoint structure.

For this purpose we intend to confine the formalisation to the part of a viewpoint which involves operators acting on a model to augment it or to draw inferences. That is we shall follow the aims of the study in not considering the heuristics or algorithms which denote a particular model as being relevant to a particular context or goal.

Although these heuristic elements are not addressed in the formalisation work we will now give brief examples of them in relation to the structural and functional models of the study as this may help to clarify subsequent discussion. The heuristics are intended to specify the areas of application for each viewpoint. (As stated above viewpoint is defined as a model combined with a set of operators which draw inferences from it, and heuristics which describe the area of application of the combined model and operators).

Simple examples of the heuristics for the structural and functional *viewpoints* are thus:

a) Where the goal is to operate the system within specific constraints, or to tutor such operation, use the functional viewpoint.

b) Where the goal is to find faults in the system, or to tutor such fault-finding, use the structural viewpoint.

5.2.3 Formalising the reasoning of the 'functional model' group.

We may reconsider the example given in chapter 4 where subjects given a functional model find themselves with a low furnace temperature. We described this as 'goal-action' reasoning. The relevant goal has only to be identified in the original model, and the corresponding actions taken. Our example of a functional model group reasoning in relation to the damping control was

S1. "Shall we reduce damping?"
S2. "To increase the furnace temperature, yes, and maybe a lot."

If the goal here is to increase the furnace temperature then what is required in order to formalise the reasoning is an operator which will retrieve any action or actions associated with that goal in the formalised model. As this operator simply accesses information that is explicit in the model, it will be referred to as the 'access operator'.

If the goal is represented as 'increase(furnace_temperature)' we may retrieve all possible actions associated with that goal using a Prolog predicate such as:

```
get_possible_actions(Ghead, List):-
        findall((Gbody, Vars), (clause(Ghead, Gbody, Vars)), List).
```

If the goal given is represented in the model as a Prolog fact, then the 'Gbody' returned will be the atom 'true'. This predicate will work for both Functional and Structural models. The operator may be characterised as enabling the straightforward retrieval of information represented in the model.

We also gave examples above of 'condition-action' reasoning by the group given a functional model where they were responding to a low power-output. If we assume that this is a response to the instruction that the system must be kept within the stated constraints, then the condition to which they are reacting is defined by reference to those constraints. The formalisation of action choice has been described above, but formalising the perception of the condition requires a different kind of operator. This is because the information required to make a decision about constraint satisfaction is not contained in a single element of the formalised models. The success of the 'inconstraints' predicate of both models requires the satisfaction of subgoals concerning ' legal_poweroutput ' and 'legal_furnacetemp'. The conditions for satisfaction of these subgoals are defined elsewhere. What is required is an operator which will chain through the different parts of the model to obtain the final numerical definition of 'inconstraints'. This may be loosely described as using an inference procedure to make explicit information which is only implicit in the model. This may be represented by a predicate such as:

```
infer_info(Goal, Result):-
        clause(Goal, Implication),
        Implication =.. [Op|Goallist],
        get_subgoals(Goallist, Result).

get_subgoals([], []).
get_subgoals([[]], []).
get_subgoals([Head|Rest], [Subgoals|Result]):-
        clause(Head, Subgoals),
        get_subgoals(Rest, Result).
```

This would assemble a list of the precise subgoals, (of subgoals), which would have to be satisfied for the initial goal of 'inconstraints(X, Y)'. An alternative example of this type of operator is given below with reference to the structural model group, where it would be

more generally used. Due to its use of an inference procedure, this operator will be referred to as the 'inference operator'.

5.2.4 Formalising the reasoning of the 'structural model' group.

The reasoning of subjects given a structural model is generally more complex, and it is necessary for them to augment the model they were given with some assumptions. Our example shows them faced with a rising furnace temperature.

"It's jumped up again, it wants more damping. Slow down the fission reaction".

This example involves firstly an assumption that the furnace temperature is dependent on the state of the reaction, and secondly an assumption that the "control" of the "damping material" over the "reaction" conforms to some model of damping that the subjects already possess. These are essentially assumptions of causality, an increase in the reaction level causing an increase in the furnace temperature etc. Apart from the model in their instructions the only explicitly relevant information available to the subjects is the readout of the furnace temperature, and the damping setting on the screen. What needs to be formalised is the reasoning which links the two items of information.

This may be done by adding Prolog clauses to the structural model formalised above indicating that one of the operators we need is an operator which will augment the existing model with new information. Although this involves the making of assumptions we do not intend to imply that all parts of the augmented model should be consistent, or 'true' in relation to each other. Instead, the formalisation is intended to be much closer to the "distributed models" described by de Sessa (1986).

In the current example the furnace temperature is too high, so the subjects have a new goal which is not represented in their original model: ' decrease(furnace_temperature)'. This goal requires an appropriate action on the system, which may be represented as:

```
produces( Change( State ), decrease( furnace_temperature )).
```

(Functors may be variables in LPA MacPROLOG version 3.0). The structural model given to the subjects does not specifically tell them that the reaction produces heat, nor that it is this heat which is one determinant of the furnace temperature. The assumption that the furnace temperature is related to the heat produced by the reaction, and the nature of this relationship, may be formalised as:

```
related( furnace_temperature, heat ).
related( heat, reaction ).
relationship( increase( reaction ), increase( heat ) ).
relationship( decrease( reaction ), decrease( heat ) ).
relationship( increase( heat ), increase( furnace_temperature ) ).
relationship( decrease( heat ), decrease( furnace_temperature ) ).
```

A fully runnable program requires additional clauses telling the Prolog interpreter how to use this representation to set up new goals on the path to satisfying the initial goal. These could be written as:

```
produces( Change( State ), Otherchange( Otherstate )):-
        involved( Otherstate, State ),
        causal( Change( State ), Otherchange( Otherstate)).
involved( W, Z ):- related( W, Y ), related( Y, Z ).
involved( W, Z ):- related(W, Z ).
causal(X, Y):-  relationship(X, Y).
causal(X, Y):-  relationship(X, Z), relationship(Z, Y).
```

This ' produces ' predicate is able to chain through the new database clauses and produce inferences which are not explicitly represented in the model, and constitutes the second

example of the 'chaining' operator described above. It is able to do so without using the 'clause' predicate, since the clauses it operates on are in the form of Prolog facts.

The second assumption in our example tells us that the level of reaction is related to the amount of damping being applied to the system. "Damping" may be modelled as a value change which is in an inverse relationship to the values of any variables it effects. Thus if the value associated with damping increases, the values of any variables it effects must decrease.

```
related( reaction, damping ).
relationship( increase( damping ), decrease( reaction ) ).
relationship( decreases( damping ), increases( reaction ) ).
```

The Prolog interpreter can exploit this code using the same interpretive clauses, (the 'produces' predicate), given above.

The control actions available to the system may be represented as Prolog "facts", such as:

```
increase( damping, N, Direction ).
decrease( damping, N, Direction ).
```

where 'N' is a variable indicating the degree of change in the current value of damping, and 'Direction' is a variable indicating the direction of that change. Formalising the execution of the action to satisfy the derived goal, would require further predicates to determine the degree of change, to calculate the new variable value, to set the execution as a goal, to assert the new value in the database, and to adjust the system to the change. This augmented model thus allows a succession of goal states to be defined, linking the top-level goal, a reduction of the furnace temperature, with the action of increasing the damping. Through the application of 'retrieval' and 'chaining' operators to an augmented model we are able to formalise how subjects who were given a structural model develop inferences

equivalent to those found in the functional model. This does not imply that their inferences are correct, consistent, or identical with those used by other subjects or groups.

5.2.5 Conclusions: formalising the reasoning of the two groups.

The work described in sections 5.2.1 - 5.2.4 leads us to claim that the most distinct differences of reasoning between the two groups in the study (ie. the use of causal as opposed to rule-based reasoning) can be formalised using the combination of models and operators which we have outlined above as a structure for implementing viewpoints. The detail of this exercise indicates that, in this case, at least three of different classes of operator are required in order to infer information from the given models or to change them. These three classes are described below.

The assertion that heuristics may be included as a part of a viewpoint in order to encode information about the problems that the viewpoint may be applied to and the goals that it may serve was not tested in the protocol analysis or in the subsequent formalisation. No claims may thus be made in relation to this topic.

5.3 Classes of operators required to formalise the reasoning.

5.3.1 Introduction.

These example formalisations indicate that, in this case, at least three classes of operator are needed if viewpoints are to be implemented in the form described above. Since our examples may be said to rely on the Prolog interpreter we shall try to characterise the three classes of operator in abstract terms so that they may be implemented in whatever language or representational formalism is considered appropriate.

5.3.2 Operator type one: the 'access' operator.

This is intended to retrieve the smallest elements of information already directly available in the model; eg. in relation to the formalised structural model above it would answer the question "What controls the steam?". What is retrieved will thus depend on the grain-size and content of the knowledge representation. Where for example the elements of the knowledge representation were binary relations, such as 'turns(A, B)', or implications such as 'A -> B', the first type of operator would simply retrieve these. This operator would be used to formalise the reasoning of both functional and structural groups, and is exemplified (in predicate form) by the 'get_possible_actions' predicate given in section 5.2.3. This predicate retrieves the second half of a binary element, or an implication relation, if one half is given.

The need for such a 'retrieval' operator is specific to this project of implementing viewpoints. Such an operator is not required for traditional problem solving systems as they are taken as having perfect access to the solution if it is a part of the goal state. In other words if the "solution" is a part of the current state, then no specific retrieval operator is required. This assumption cannot be made in relation to viewpoints as we may wish, for tutoring purposes, to ask two different questions of the same representation; (cf. the different uses of the same circuit representation in SOPHIE I [Brown and Burton 1975]). We may wish the single structural model (formalised above) to tell us firstly: 'What moves the coolant?' or on another occasion: 'What controls the steam?'

5.3.3 Operator type two: the 'inference' operator.

This operator examines the model, and combines two or more parts of it via an inference procedure to produce a previously unstated conclusion. It does not, however, introduce new information from outside the original model; (this is done by the third type of operator). There may be as many examples of the 'inference' operator as there are

inference procedures. The 'infer_info' and ' produces ' predicates given in section 5.2.3 are examples in predicate form. 'Infer_info' links a goal state with the stated actions necessary to achieve it, while 'produces' is used in the example given to determine the action which will reduce the furnace temperature. 'Produces' does this by defining an inference procedure for specific binary relations, in terms of legal relationships between the functors and their arguments. This type of operator could thus be to formalise the reasoning of both functional and structural groups.

A more general example of the operator could be a rule of inference in logic, where:

(A -> B, B -> C) -> (A -> C).

5.3.4 Operator Type Three.

This operator directly introduces new information which is related to the system state. This operator is exemplified by the last example formalised above, where models of reaction and damping were imported and added to the original model. It is assumed that this operator would be most generally used in relation to the group given a structural model.

The added models are not derived from the original model but are assumed to exist separately as the "real-world knowledge" of the person whose reasoning is being formalised. The result is a composite model to which the first and second classes of operators described above may be applied. We view these as being similar to the "distributed" models described by di Sessa, which are "... accumulated from multiple, partial, explanations ...", and which "... represent a patchwork collection of pre-existing ideas in the learner, 'corrupted' to new ends." (di Sessa [1986] p. 209). This may be seen as the union of two sets of elements, and does not require that the result be coherent. Our current work assumes that a range of possible parts to be added would have been pre-defined by the system designer. The operator is thus not the parts that are added, but the mechanism that adds them. The literature describes a number of mechanisms for

determining which information should be retrieved and drawn into the situation, such as Clancey's (1985) 'Heuristic Classification'.

5.4 Conclusions to Chapter 5.

As noted previously, the goals of the study described in chapter 4 and the formalisation described in chapter 5 were to identify different reasoning patterns associated with the use of different models, so that these differences could be formalised using the proposed conceptual structure for implementing viewpoints in an ITS; (see chapter 3). This structure consists of a model, a set of operators which draw inferences from it, and a set of heuristics which indicate the viewpoint's area of application. The distinct reasoning patterns identified in chapter 4 were formalised in order to demonstrate the utility of the proposed viewpoint structure, and to clarify the nature of the operators that would be required to draw inferences from the models in an implementation.

Examples of the different reasoning patterns drawn from the protocols of chapter 4 were formalised in Prolog code using a Prolog formalisation of the models given to the subjects, and operators developed in response to the needs of the exercise. This led to the identification of three distinct classes of operators which may be implemented in any suitable formalism. It is thus suggested that both of the goals of the study have been realised: ie. the utility of the proposed structure for viewpoints has been demonstrated, and greater knowledge gained of the kinds of operator which are required in order to implement it.

Stronger conclusions concerning the different inference processes involved in the use of different models may not be drawn, due to the limited numbers in each group of subjects. Also, no claims are made as to the precise psychological mechanism, (eg. analogy, qualitative reasoning), involved in the manipulation of the models described. The question of heuristics to select a given viewpoint as appropriate to a specific problem has not been investigated here.

Chapter 6. The Implementation Domain and Tutoring Goals.

6.1 Introduction.

Previous chapters have introduced the area to be investigated in this thesis, (the use of multiple viewpoints in ITS), and described the development and validation of a structure which may be used to formalise viewpoints in preparation for their implementation in a tutoring system. This chapter deals specifically with the formulation of the domain which is to be tutored.

Having verified the usefulness of the viewpoint structure it was necessary to choose an example domain in which to implement a tutoring system. A very strong motivation at this point was that a *real* domain should be chosen. By this we mean a domain where the actual viewpoints used by practitioners or students in that domain could be formalised and used and some attempt made to deal with the real problems and misconceptions that they faced. There were several reasons for this motivation. The first and most general was a belief that the appropriate way for tutoring systems technology to advance was for it to engage with real rather than 'toy' domains, and their related problems. The second reason related to the intended evaluation of the implemented system. If a fictional domain, or an excessively arcane one were to be chosen, then serious problems could arise in finding a sufficiently large user population to conduct some form of evaluation. A third reason was the simple wish that the work done should at least have the chance of being of some practical use when it was completed. It was felt that the issues involved were of considerable significance to system design and to education generally, and that every attempt should be made to demonstrate this to those who were not yet convinced.

These considerations meant that the domain used for the study described in chapter 4 (nuclear reactor/power station) was not suitable. True, there is a pressing need for reliable and effective training to be given to the operators of such systems in the real world, but the

nature of these systems and the complexity of the knowledge required to deal with them meant that the accurate acquisition and representation of such a domain was a project which exceeded the resources available for completion of the thesis. It was also foreseen that there would be considerable problems in finding a suitable pool of practitioners who could take part in an evaluation.

What was required was a 'real' domain of suitable size where the relevant viewpoints had at least been identified, where a definite educational need had been established, and where a sufficient number of users could be found to conduct an evaluation after the system had been implemented. These needs were answered by the domain of Prolog, especially of Prolog for novices. The language is frequently a mystery to those who are encountering it for the first time, and sometimes even to those who are more experienced in its use. Attempts to alleviate this situation have included the description of a series of models (Bundy et al. 1985) which may be taught to novices so as to give them a more structured initial understanding of the language. We assumed that viewpoints based on these models could be developed. As the structure used to define viewpoints emphasises the use of the knowledge involved, we wished to define a domain where the different viewpoints could actually be applied, and where such application could be practised and critiqued as a part of the tutoring process. Models of execution are clearly necessary in the task of debugging. If students have problems understanding how Prolog works, then they will have even greater problems in debugging code. These considerations led to the choice of Prolog debugging for novices as the experimental application domain. The goal was to build a system which could tutor the skill of using different viewpoints to localise bugs in Prolog code.

This domain is summarised as the use by novices of different viewpoints on program execution in order to localise bugs in Prolog code. Two partial accounts of the domain have to be formulated and combined in order to produce a set of abstractions which, when implemented, can act as the knowledge representations of the tutoring system.

The two partial accounts of the domain are:

1) Three complementary models of Prolog execution; (based on a set of *four* models developed by Bundy et al. [1985]).. These describe a subset of Prolog behaviour, and are intended to support the tutoring of novices. When combined with operators and heuristics as described in chapter 3, each model forms part of a viewpoint.

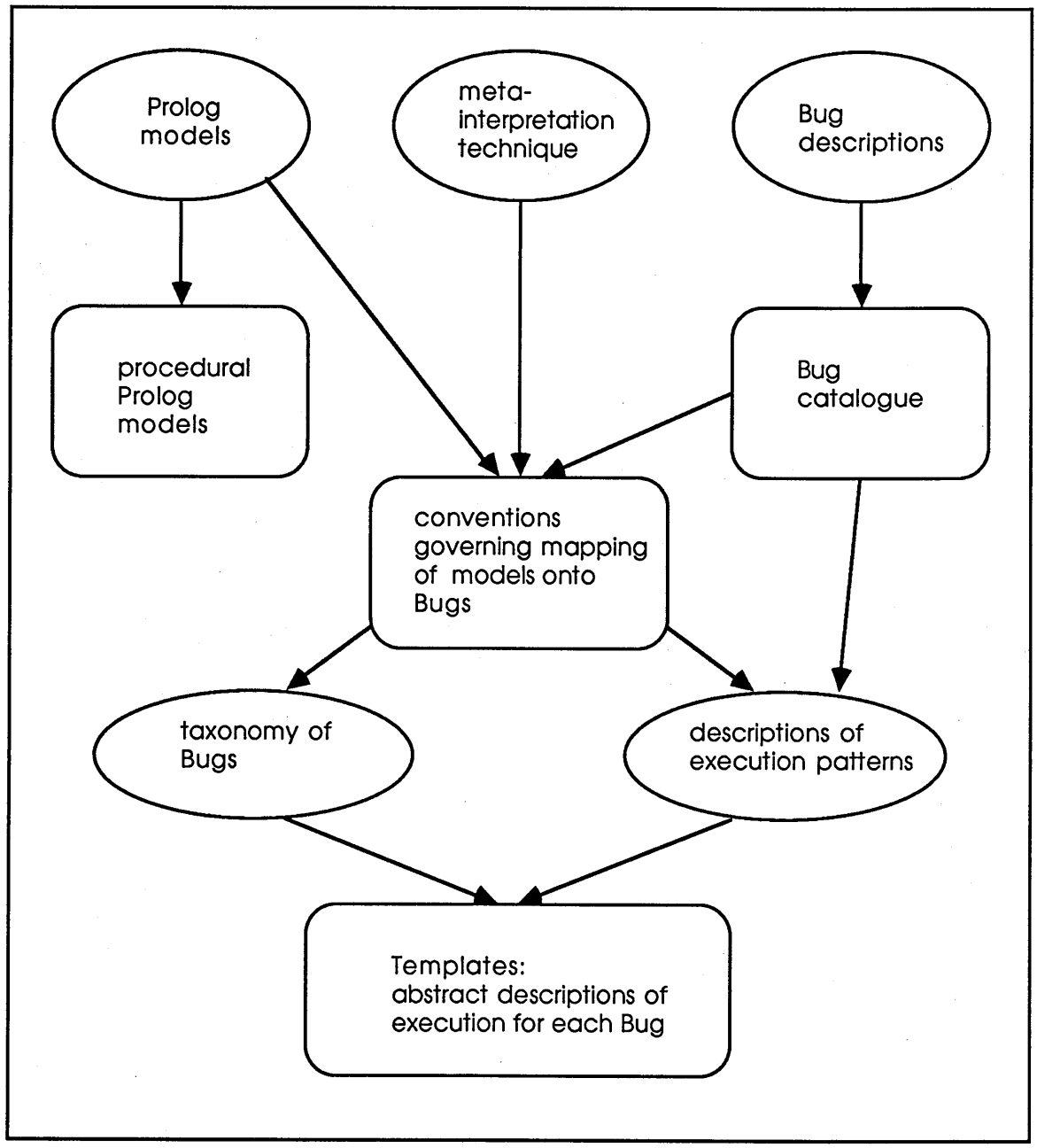
2) A restricted catalogue of bugs. Bugs are described syntactically in terms of 'missing', 'extra', or 'wrong' 'modules'. These descriptions are also intended to support the tutoring of novices.

The work reported in this chapter is described graphically in figure 4.

Figure 4 depicts the process by which the models of Prolog and the catalogue of bugs are combined. First, the possible execution patterns for each bug are listed. Then a set of conventions is established which governs the mapping of the models onto the bugs. This set of conventions is used to define a taxonomy of bugs and to describe the effects of each bug on program execution. These conventions also take account of the intended use of a particular technique for interpreting a record of Prolog execution. The final abstractions that the tutoring system uses are produced by combining the descriptions of each bug's effect on program execution with a statement of the bug's classification.

To avoid confusion it should be noted that while Bundy et al. (1985) describe four models of Prolog, the work described in this chapter is based on only three of these, which are re-defined for specific uses in the tutoring system.

Figure 4. An overall view of the work described in chapter 6.



As previously stated, both the execution models and the bug catalogue are intended to support the tutoring of novices. To this end, both are simplified accounts of their subjects. The Prolog models developed in this chapter only deal with a subset of Prolog execution, and the bug catalogue only describes bugs in syntactic terms, (rather than in terms of the programmers intentions or specific programming techniques). The models and bug

catalogue thus constitute two simplified environments which are suited to the needs of novices. The novices can explore the principles of the domain without necessarily acquiring the detailed knowledge used at a more advanced stage, and without being distracted by details or discouraged by their lack of knowledge.

It is intended that the partial accounts of the domain should be upwardly compatible with more complete accounts which could follow. An earlier example of this simplification strategy being used in an ITS is the succession of augmented models in QUEST (White and Frederiksen 1986).

6.2 Prolog for novices.

6.2.1 Introduction.

The choice of a tutoring system's domain is an integral part of the design process, as is the precise formulation of that domain for implementation in the system's knowledge base. The work described in this section is motivated by the need to describe Prolog execution in terms of models which are accessible to novices. A more precise statement of that motivation requires that several topics which have been introduced separately should now be considered in relation to each other.

In section 2.3.1, four models which were developed by Bundy et al. (1985) for describing Prolog execution to novices were introduced; (the rest of the chapter concentrates on three of these). A technique developed by Eisenstadt (1984) for interpreting a history of Prolog execution was introduced in section 2.3.2. In section 3.3.2 a possible system was discussed based on the assumptions that its domain was the localisation of bugs in Prolog code, and that a number of implemented viewpoints were available to it which together could describe Prolog execution. The purpose of the assumed system was to tutor the skill of using the viewpoints to localise bugs in pieces of code.

The implementation of such a system requires that the execution of the code being considered by the student should be interpreted in terms of the relevant viewpoints. Such an interpretation would enable the critiquing of reasoning about the execution by both system and student. In the context of this thesis, this implies that the complementary models of Prolog execution described by Bundy et al. (1985) should be formulated in such a way that they may act as the basis of viewpoints in the implemented tutoring system. Specifically, they should be formulated in a way which allows them to be combined to form a working and observable Prolog interpreter. A record of the workings of this interpreter could then be interpreted in the manner described by Eisenstadt (1984).

This section thus describes the formulation of the models from Bundy et al. (1985) to serve as components of such an interpreter. Some parts of their models, such as backtracking and the use of the cut, are omitted. Other parts, such as the generation and processing of subgoals, are changed.

6.2.2 Background to the models of Prolog execution.

As described in chapter 2, the difficulties encountered by Prolog novices motivated Pain and Bundy (1985) and Bundy et al. (1985) to produce a complete and consistent "Prolog story" which could be used to "...understand and predict the execution of a Prolog program.", and which could form the basis for teaching materials, error messages, and tracing packages. The "story" was intended to cover both the procedural and declarative semantics of Prolog, and especially illuminate the 'difficult to understand' aspects of the language such as the construction of recursive data structures and the scope of variables. This required that the details of resolutions, the proof of outstanding goals, and the search strategy employed should all be described. "Impurities" in this story are described by Brna et al. (1987b).

As stated above, the resultant story has four parts, which we shall regard as complementary models in the sense of the "user's conceptual model" discussed by Young (1983). The purpose of such 'conceptual models' is to allow the user to make reliable inferences about the system being considered. The parts of Bundy et al.'s (1985) 'story' are the Program Database, the Search Space, the Search Strategy, and the Resolution Process. The short overview of these models presented in chapter 2 is now repeated in the next paragraph as an introduction to a more detailed discussion.

In various combinations, the models can be used to comprehend the many different representations of Prolog execution, (Byrd Boxes, Arrow Diagrams, And/Or Trees, Or Trees, Full Trace, Partial Trace etc.). The Database is the collection of assertion and implication clauses that go to make up the Prolog program. The Search Space describes the relationship between the goal literals which are input by the user, or generated by the program, and the Program Database. The Search Strategy is concerned with the order in which the goal literals are generated, and the order in which Database clauses are chosen for resolution with them. The Resolution Process describes the unification of goal and clausehead, the binding of variables to values, and the possible generation of new goal literals.

Bundy et al. (1985) do not give a detailed statement of each model, but instead give a short outline and examples of its application. A preliminary task was thus the formulation of some more precise statement of each model. The initial attempt at this resulted in the model statements given below. These were later modified to facilitate the implementation of the tutoring system.

Each model was first defined in detail as a set of Prolog clauses. The form given here is that of the Prolog clauses translated into English sentences.

Before examining these models it may be helpful to describe their relationship to the "viewpoints" of the implemented tutoring system. This relationship is outlined in the introduction to this chapter, where the process of formulating an account of the system's domain is summarised. The process is descibed graphically in figure 4. The system domain is summarised as the "... use by novices of different viewpoints on program execution to localise bugs in Prolog code", the point of this being that different viewpoints may be used to localise different classes of bugs. In order to formulate the domain, two partial accounts have to be combined in a way which facilitates the description of bugged execution in terms of the viewpoints we wish to use. As described in section 6.1 these two partial accounts are 1) a set of models which describe Prolog execution, and 2) a restricted catalogue of syntactically-described bugs. The models become "viewpoints" when they are combined with operators which draw inferences from them to identify bugs, and heuristics which state the kind of problem to which the "viewpoint" can be applied.

The purpose of sections 6.2.3 - 6.2.6 is to produce a first statement of the first partial account of the domain, that of the set of models for describing Prolog execution. This statement is later modified to serve the needs of the implementation.

6.2.3 Prolog Model 1: The Database of Clauses.

1. The Database can contain Facts.
2. The Database can contain Rules.
3. The Database can contain Facts and Rules, but only Facts and Rules.
4. Facts and Rules are composed of Prolog terms.
5. A fact has no rhs. and ends in a full stop.
6. A rule has a word, or a functor, or a relationship for a lhs, ':-' as a separator, with subgoal(s) for a rhs.
7. A subgoal has the form of a fact.
8. Functors have arguments which are within brackets.
9. A fact may be a word.
10. A fact may have a functor and an argument.
11. A fact may be a relation.

12. A relation has a functor and two or more arguments.
13. Facts and Relations can be nested objects.
14. A Fact is true.
15. The head of a rule is true if the subgoals of the same rule are true.

This model was seen as being largely concerned with syntax, and, as Bundy et al. (1985) state, the possible forms of "...assertion and implication clauses... ". This is taken to mean that syntactically correct facts are true, while for syntactically correct rules, the head is true if the subgoals are true. It should be noted that no information is given about the relationships *between* the different elements of the Database (ie. rules, subgoals and facts). Thus, while this model embodies the purpose of the rule form by stating the relationship of implication between rulehead and subgoals, it does *not* tell us how to prove those subgoals.

The detailed syntax of Prolog is defined through a number of other statements which define the possible alternatives for 'term', 'word', and 'relation'.

This model, (like the other three), was implemented, but does not play any part in the final tutoring system. This is because the system is itself built in Prolog, (ie. the models of Prolog are implemented in Prolog code), and syntax checking is done automatically by the host environment before any of the code is executed. Since it is the execution of Prolog, rather that its syntax which is expected to give novices problems, a decision was taken to concentrate on the other three models. The model was implemented to the point where it could accurately diagnose and label correct and incorrect Prolog terms.

6.2.4 Prolog Model 2: The Search Space.

If this model is to be used without reference to model 1, then the following three statements concerning the possible contents of the Search Space are necessary as the Search Space is concerned with the relationships between 'facts' and 'rules':

1. The Search Space can contain Facts.
2. The Search Space can contain Rules.
3. The Search Space can contain Facts and Rules, but only Facts and Rules.

The question of whether these statements are more properly a part of the Syntax model is ultimately academic, as the operation of the Search Space model in the tutoring system does not refer directly to these three statements, but must have them as a prerequisite assumption.

The main part of the model is as follows:

1. A goal is true if it resolves with a fact or if it resolves with the head of a rule whose subgoals can be shown to be true using the contents of the Search Space.
2. Subgoals are treated as goals.
3. The head of a rule is true if its subgoal(s) is/are true as defined in 1 and 2.
4. There may be more than one way of showing a goal to be true.

These statements may be paraphrased as describing the space of facts and rules which is available for proving goals, and the relationships between the parts, (facts and rules), of that space. This may be likened to normal theorem-proving, where the successive goals and theorems to be considered are not implicitly ordered. It is the stated relationship between the facts and rules which distinguishes this model from the 'Database' model. This model can thus be seen as a 'declarative' statement of the knowledge needed to determine the relationship between the rules referred to in the Database model. It may be labelled 'declarative', as it states the conditions which are to determine the truth or failure of a given goal, but gives no information about *how* any search to determine that truth or failure is to be carried out. In other words, it defines a structure without giving any information about how to move around it. This information about *how* to carry out the search may be labelled 'procedural' information, and is the essence of the third model, the Search Strategy.

The point of describing the Search Space in these terms (which are not found in the work of Bundy et al. [1985]) is that they provide an abstract definition of the concept of "Search Space". This abstract definition can then be used to draw inferences about all instances of that concept, ie. about any specific Search Space that is encountered. The more usual definitions of a Search Space in terms of 'Call Graphs', 'OR Trees', or 'AND/OR Trees' always have their structure and relationships dictated by the specific Seach Space being described, and are thus a descriptions of a specific instance of a Search Space which cannot be used to draw deductive inferences about any other Search Space. If a viewpoint is to be useful, it must be based on a model which has a suitable degreee of generality. The model of the Search Space has thus been formulated to give this generality.

6.2.5 Prolog Model 3: The Search Strategy.

As indicated in section 6.2.4, this model gives the 'procedural' knowledge required to organise a search through the Search Space in a particular uniform and consistent way. This is contrasted with the declarative knowledge given in the Search Space model described in the last section. This model specifies the order in which rules and facts are to be considered as parts of possible proofs for a given goal, and the order in which any resultant subgoals are themselves to be considered for proof. The model's statements use the verbs 'resolve' and 'unify'. These actions are not defined in this model, but form the core of the fourth model: Resolution.

1. To show that a goal is true, attempt to resolve the goal/query with successive database items in the order in which the items occur in the database.

1a. If the goal resolves with the item and the item is a fact, then the goal is true.

1b. If the goal matches the head of a rule, then the goal is true if the subgoals of the rule can be shown to be true, taken in the order in which they occur in the rule.

1c. If the current attempt to prove a goal true fails, then attempt to resolve the goal with the remaining items in the database, in the order in which they occur there. If a match is found, repeat 1a/1b.

2. If no fact resolves with the goal, or if no set of subgoals for a matching rulehead can be shown to be true, then the goal is not true.

Two points can be noted here: firstly that this model makes no reference to backtracking as an element of the Search Strategy, and secondly that the generation of new goal literals from the subgoals of a rule *is* included as a part of the Search Strategy.

Backtracking was excluded in the prototype version of the system, as our intention was to use, in the first instance, examples which did not depend on backtracking to produce their results. It was envisaged that a hierarchy of models could be defined which described Prolog execution at increasing levels of complexity until the full range of Prolog behaviour had been covered. It was assumed that, given sufficient time, the prototype system could be expanded to cater for all of these models and their combination with operators and heuristics to form viewpoints.

This strategy of starting with simple models and progressing to more complex ones has precedents in the ITS literature, and underlies the design of QUEST (White and Frederiksen 1986). If such a succession of models was desired for the Prolog story, with backtracking included in the more complex ones, this could be achieved by expanding item 2 of the model given above to include the requisite information about re-doing the most recent successful goal if the current one fails. This would also satisfy White and Fredriksen's criteria of "upward compatibility". (This requires that a student should be able to refine and extend the models used at an early stage of a learning process with a minimum of reconceptualisation). This expansion would however, also occasion some extra complexity in the 'Resolution' model given below, since this would require a

description of how variables are unbound if backtracking proceeds through the resolution that initially bound them.

It was noted above that the generation of new goal literals from subgoals is here described as an aspect of the Search Strategy. This is a departure from the original 'story' of Bundy et al. (1985), where the formation of new goal literals is described as an aspect of Resolution. The change was made in the interests of consistency and economy. It is clear that the proving of subgoals has to be somehow connected to the description of the Search Space and Search Strategy, since subgoals must be proved in the Search Space, using the same strategy as for an initial query. It would hardly be economical to have the same search strategy described twice, ie. once for initial queries whose proof was described in the Search Strategy model, and again for subgoals whose proof was described as a part of the Resolution model. Such a dual description would also fail to promote an appreciation of the recursive nature of Prolog. Accordingly, the Search Space model declares that subgoals are to be treated as goals, ie. that the same Search Strategy should be applied to proving them. The Search Strategy model itself thus regards subgoals and queries as in some senses equivalent.

There is another reason for constructing the models in this way. This relates to the fact that there is a strategy governing the *order* in which the subgoals are proved, ie. they are attempted in the order in which they appear in the rule rather than in any random order. This point is not emphasised in Bundy et al. (1985), although they refer to it in their discussion of the way in which the Search Strategy imposes an order on the nodes of an Execution tree. (This implies that in describing a given execution, the nodes of the tree would conventionally be visited in a top-down, left-to-right order). There are thus two aspects of search strategy which affect the handling of subgoals. Firstly, they are proved in the same manner as an input query. Secondly, where there is more than one subgoal, the goals should be proved in the order in which they appear in the rule. It was decided

that each of these issues could be made more clearly explicit by removing the handling of subgoals from the Resolution model, and including it in the Search Strategy model.

6.2.6 Prolog Model 4: The Resolution Process.

The model below might more accurately be called the 'unification' process, since, unlike the 'Resolution' story proposed by Bundy et al. (1985) it does not deal with the generation of subgoals. For the reasons given in the last section, the generation of subgoals is described as a part of the Search Strategy. The process described here is the unification of a goal with the head of a database item. If that item happens to be a fact, (ie. it has no subgoals), then this unification is equivalent to a resolution in the sense that the goal is proved. If there *are* subgoals to be proved, then the process described in this model will not complete the resolution, as the Search Strategy will have to be employed to set the subgoals as new goal literals. The model reads as follows:

1. Two literals will unify if:

   a) Their principal functors are the same

   AND

   b) They have the same number of arguments, (if any)

   AND

   when each argument in one goal is matched with the corresponding argument in the same position in the other goal, starting with the leftmost,

   c) *Either*

      1) The arguments are terms which are identical

      OR

      2) Both arguments are variables which are either uninstantiated or instantiated to the same value.

      OR

      3) An uninstantiated variable in one clause is in the same position as a Prolog term in the other.

      AND

      all occurrences of a given variable in the same two goal literals can be instantiated to the same value without contradicting another argument (or instantiation).

If these conditions are satisfied, the variable is instantiated to the value of the Prolog term.

2. If these conditions are not satisfied, the goal literals will not unify, and any variable instantiations made in this resolution are abandoned.

In the quest for clarity it was decided to keep this model as simple as possible in order to facilitate its acquisition and use by novices. To this end, the smallest unit of description that this model recognises is an 'argument' which is instantiated as a 'term' or a 'variable'. The effect of this is that the model cannot, immediately, deal with embedded variables or "functions" in terms (ie. in arguments). In this sense the model describes a "function-free" Prolog. It is assumed that the model could be expanded to a more complex form which was able to deal with such functons. As was argued above in relation to backtracking, this simplification is consistent with the pedagogical practice of providing a sequence of models with increasing levels of complexity, and beginning the tutoring with the simplest.

The model given here for 'resolution' *could* deal with embedded variables if the part of it which describes argument matching was applied recursively to the elements of each term which constituted the arguments; (ie. each bit of the two arguments being unified had to either be identical, or a variable ... etc.). This would describe the unification of arguments at a finer grain size, and could constitute the next level of complexity to be tutored if a progression of models was desired. The model as it is given above is sufficient for our current purposes.

6.2.7 The models combined as an interpreter.

This chapter is mainly concerned with defining the domain in which the tutoring system is to operate. As the opening remarks of this section point out, however, one goal of the formulation of the models is that they should, in combination, form a Prolog interpreter whose operation is transparent in the sense that it can be observed. This is equivalent to the "glass box" representation described in section 2.1.4. Since our adapted definitions of the

models have just been discussed, this is a suitable point at which to note some of their shortcomings when used as the basis for such a glass-box interpreter. These shortcomings do not relate to specific details of the implementation, but to aspects of Prolog execution which anything claiming to be an interpreter would have to demonstrate.

The chief shortcoming of the models, assuming that they are to be combined to form a Prolog interpreter, is that there is no account of a) the way in which variables are standardised apart, and b) the method by which instantiated values are passed up and down between parent goals and subgoals. The first issue, the standardisation apart of differently-named variables, describes the process by which unbound variables are given a new unique identifier when a goal containing them becomes a new goal literal. This issue is mentioned briefly by Bundy et al. (1985) as an aspect of their 'Resolution' story, although no details of the rules governing the process are given.

The second issue, that of the method by which instantiated values are passed up and down between parent goals and subgoals, is illustrated briefly by Bundy et. al, although they do not describe how the process is accomplished. It may well be a matter for debate whether this issue, the passing of variable values, is properly discussed as an aspect of the 'pure' declarative account of Prolog, or as an aspect of the implementations of the specified language. The latter category would imply that the issue was not crucial to understanding the language, and thus not of great interest in the tutoring of debugging strategies to novices. The view taken here is that neither of these issues is of immediate relevance to novices, and thus need not be pursued any further.

6.2.8 Conclusions to section 6.2

Four complementary models of Prolog execution have been formulated. Three of these models are to be used as the basis of viewpoints on Prolog execution in the proposed tutoring system. The models are based on the four proposed by Bundy et al. (1985), although some parts of their models, such as backtracking and the use of the cut, are

omitted. Other parts, such as the generation and processing of subgoals, are changed. Since the intention is to implement the resulting models and combine them to form a working Prolog interpreter, their limitations for this purpose are noted.

6.3 Describing Bugs

The parts of the "story" detailed in section 6.2 could be applied to tutoring Prolog in many ways; (eg. they could be used to describe some arbitrary execution as opposed to specific programming techniques or algorithms). In order to focus the process of designing a system which could tutor in terms of the models, it was decided to concentrate on the area of debugging strategies for novices. This section details a method of classifying bugs, and its use to generate a space of possible bugs in which the implemented tutoring system can operate. As stated in section 6.1, this 'space of possible bugs' is intended to form a simplified environment which is suitable for use by novices. The space of bugs is described graphically as a set of 'trees'.

6.3.1 Classifying Bugs: a simplified environment.

As previously stated, it is intended that the domain of the proposed tutoring system should be debugging strategies for novices. This presents the problem of how bugs are to be described and classified. Section 2.4 summarised the four-level classification scheme proposed by Brna et al. (1987), and indicated that our current concern was with the 'Symptom Description', and the 'Program Code Error Description' levels. As explained in section 2.4 we wish to describe and classify a range of bugs without reference to the programmer's intentions, although Brna et al.'s (1987) discussion is related to programmer expectations. Instead of considering programmer expectations, we shall relate the classification to an ideal 'template' version of the code. Used in this way, the classification allows all possible bugs to be listed.

The categories listed by Brna et al. at the symptom level were Error Messages, Side Effects, Termination Issues and the Instantiation of Variables. Given the tutorial goals stated in the previous paragraph, the categories which are most suited to the construction of a tutoring domain are Termination Issues and the Instantiation of Variables, since they are most closely related to normal Prolog execution, and are most easily explained in terms of the models of section 6.2.

Termination issues consist of:

Unexpected termination.

Unexpected failure to terminate.

Termination with an unexpected value.

Instantiation of Variable issues consist of:

Unexpected failure to instantiate a variable.

Unexpected instantiation of a variable.

A variable instantiated to an unexpected value.

Section 2.4.1 described how these 'symptoms' may be explained at the 'Code Error' level through either syntactic or technique-oriented classifications. Our concern is with the syntactic classification. This explains symptoms in terms of missing, extra, or wrong 'modules'. Depending on the level of description, these 'modules' can be such entities as a set of predicates, or within a clause, an argument or subgoal. As stated, the search strategy of Prolog requires that we also include the possibility of wrong order for clauses and subgoals; (the topic of order is discussed by Brna in relation to both Program Misbehaviour and Code Error description). A complete list of modules allows all possible (individual) bugs to be described in relation to the ideal template code.

The modules chosen for the novice-tutoring system are as follows:

1) Clause.

2) Functor.

3) Argument.

4) Subgoal.

5) Operator.

Each of these, apart from 'Operator', can be prefixed by 'wrong', 'extra', or 'missing' to give a category of bug. To cater for bugs related to Prolog's search strategy, it is also necessary to include the categories of 'wrong order' for clauses, subgoals, and arguments. Not every combination of prefix and module name is useful; eg. the category 'wrong clause' is not useful as other descriptions such as 'wrong argument' or 'wrong argument order' are more specific. The categories of 'missing operator' or 'extra operator' seem more related to an appreciation of syntax than of execution. In all, twelve bug categories are defined. These are detailed below.

This catalogue of bugs may now be related to each of the symptoms described above. The development strategy adopted for the proposed system was to choose one of these symptoms and produce a domain representation of the bugs which could cause it with the given list of modules. It was assumed that the tutoring would be based on this domain representation, and that similar representations could be built for the other symptoms. The symptom chosen for this exercise was 'A variable instantiated to an unexpected value' as this was judged to describe an easily-comprehended situation which might well be within the student's experience, and which could be explored via a variety of tutoring activities. Accordingly, discussion in subsequent sections will generally be related to this symptom, with the exception of discussion relating to figure 5.

For each symptom, we can define a 'tree' of expected and actual outcomes and candidate bugs. As an example, the 'instantiation to an unexpected value' symptom implies that a goal containing variables succeeds in both the ideal and bugged code, and that a variable in the query is instantiated. Only bugs capable of yielding this result need to be put into the

tree. A system module can then be built to detect the bugs specified in each tree. Figure 5 gives the 'tree' for the symptom 'Unexpected Instantiation of a Variable'.

Before exploring the detail of these 'bug trees', one further point concerning their use will be made.

If three simple conditions are observed, the trees can be used to specify the range of possible bugs which may be included in the problems set to the student, and can thus form a structure for the entire domain and its tutoring. These conditions are now stated, followed by their pedagogical and technical rationale.

The conditions are as follows:

1) Only a single bug may be present in each problem.

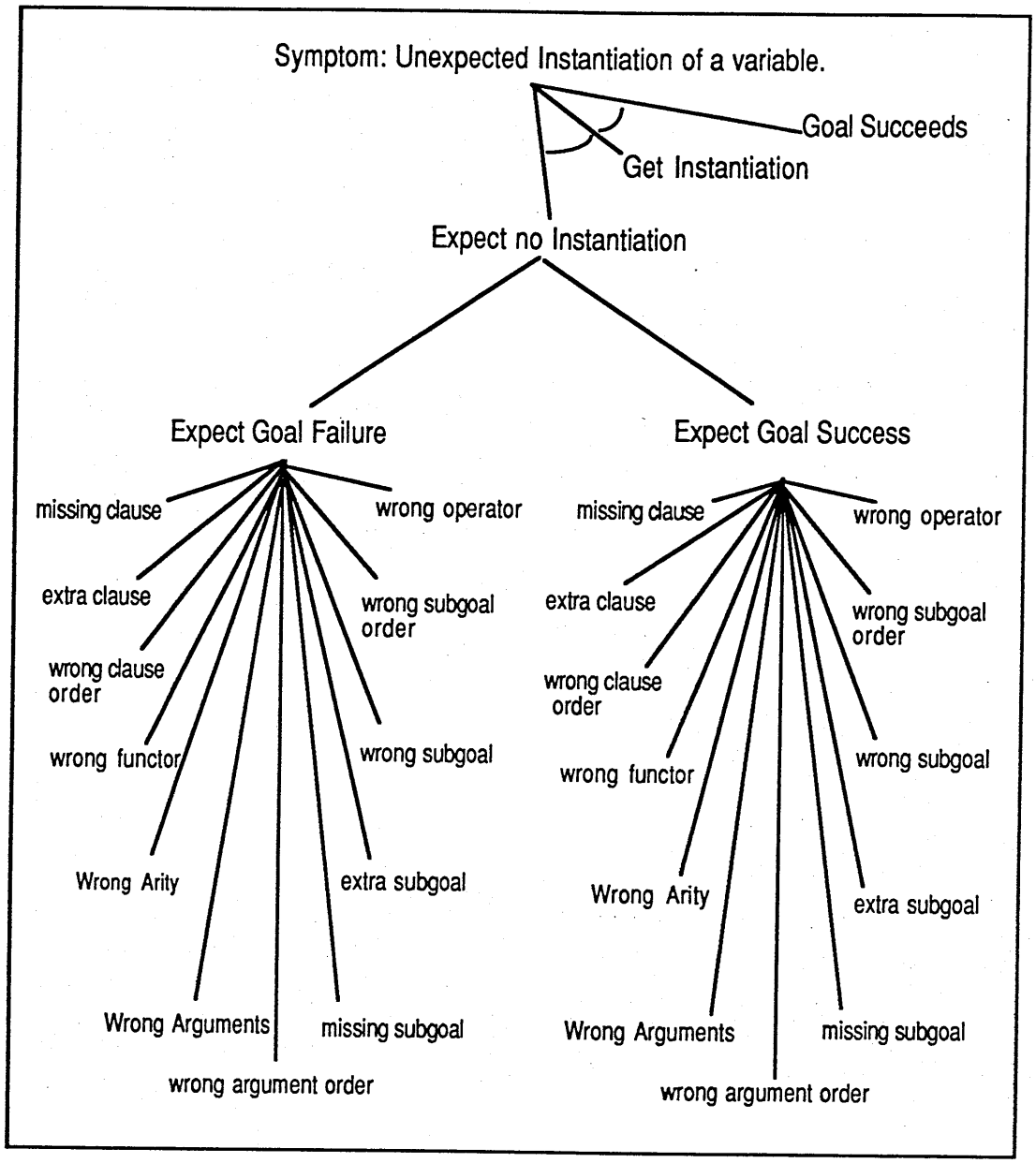
2) The bugged code may only have one difference from the ideal code.

3) The bugs should be in the databases, not in the queries set with them.

The point of these simplifying conditions is to provide an environment where the student can concentrate not simply on the localisation of bugs, but on the use of the models of Prolog developed above to carry out the task.

It is assumed that this requires, at least initially, a domain which is as simple as possible. At the same time, it is assumed that in order to tutor effectively, the system must be able to carry out the task that it is trying to tutor. Specifying that only a single bug should be present gives the simplest situation that can be described by the 'symptom and module' analysis outlined above. Multiple bugs might interact in arbitrarily complex ways, causing confusion to the student, and possibly providing patterns of execution which were beyond the capacity of the proposed system to analyse.

Figure 5. The Bug Tree for the symptom 'Unexpected instantiation of a variable'. (Symptom from Brna et al. 1987).



The stipulation that there should be only one difference between the ideal and bugged code (ie. that of the bug itself) is intended to further simplify the 'bugfinding' and explanation mechanisms of the proposed system. The stipulation allows the point of difference to be quickly identified by syntactic comparison, leaving its significance to be determined from the execution trace. As described in section 2.4, it is not the goal of this work to build a full-scale intelligent debugger. The conditions outlined here provide a closed and

structured 'world' which can be analysed by a fairly simple 'bugfinder'. This simplification allows greater effort to be directed towards the main goals of the research, ie. the development of mechanisms which describe the bug's effect on program execution in terms of the models we wish to use, and which are able to tutor this skill.

6.3.2 The Bug Trees and their application.

This section gives the 'trees' of possible bugs for the 'Variable Instantiation' symptoms given the range of 'modules' specified above. As indicated in section 6.3.1 not all combinations of 'missing', 'extra', or 'wrong' and a module are useful, and some combinations are omitted. Thus the two *possible* categories of 'missing argument' and 'extra argument' are subsumed under the single heading of 'wrong arity'. The *possible* category of 'wrong clause' is assumed to be expressed in the other, more specific, categories such as 'wrong subgoal' or 'wrong argument'. The *possible* categories 'Missing Functor' and 'Extra Functor' are not included in the domain representation. The former is judged to an unlikely event, while the latter is viewed as a syntax error. As described in section 6.3.1, the *possible* categories of 'Missing Operator' and 'Extra Operator' are also omitted. It is assumed that 'not' and 'fail' will always appear in or as subgoals.

This gives the following 12 categories of bug:

Missing Clause.

Extra Clause.

Wrong Clause Order.

Wrong Functor.

Wrong Arity.

Wrong Arguments.

Wrong Argument Order.

Missing Subgoal.

Extra Subgoal.

Wrong Subgoal.

Wrong Subgoal Order.

Wrong Operator.

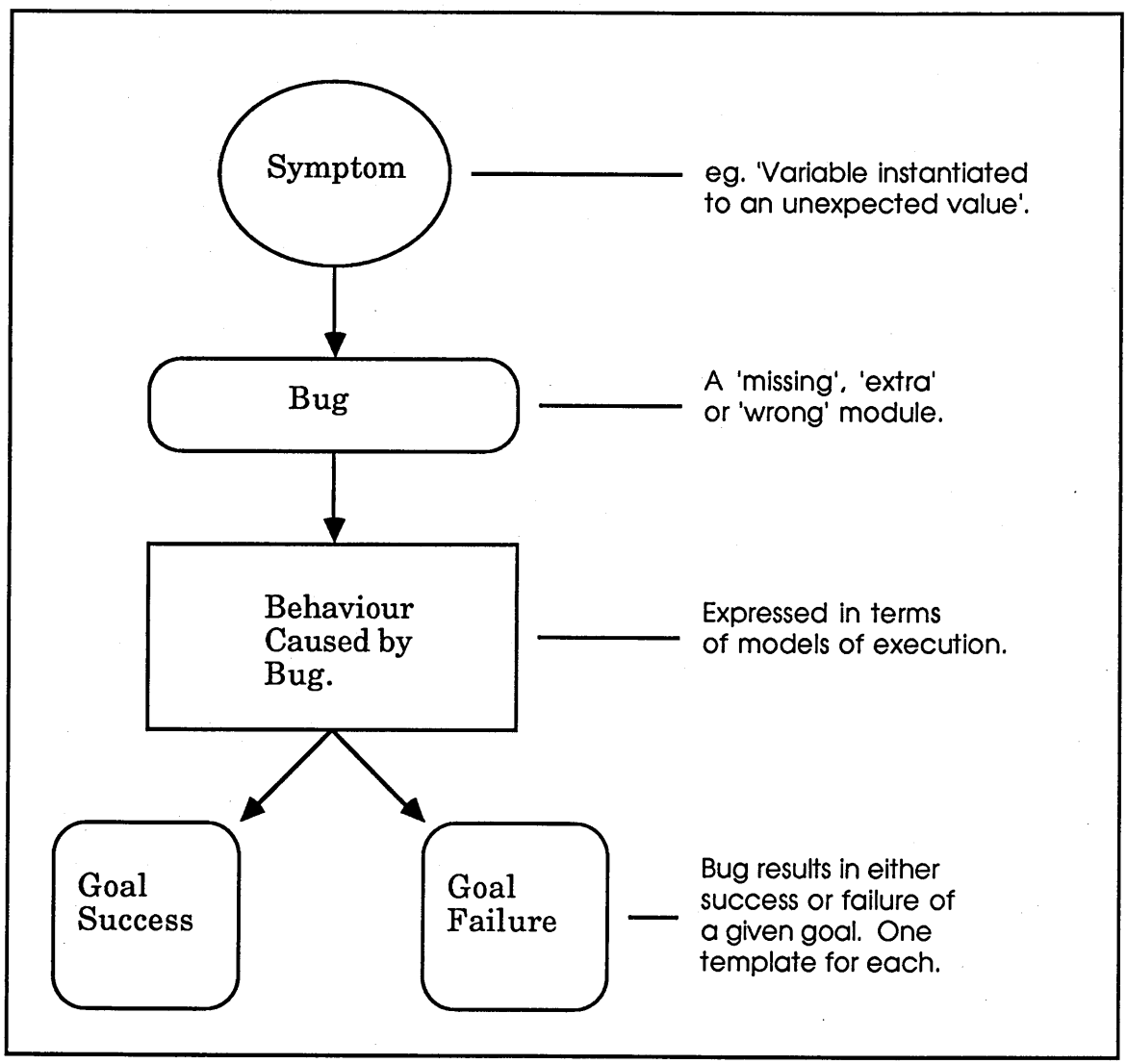
Figure 5 shows the tree which could be drawn using these categories for the symptom of 'unexpected instantiation of a variable'. Small arcs between branches of the tree indicate an 'and' relation. Absence of an arc indicates an 'or' relation. (The 'and' branch for 'goal succeeds' at the top level is something of a tautology, as in order to get the unexpected instantiation, the goal must succeed. It is included in order to establish a standard format for such trees).

As figure 5 indicates, the lack of an instantiation could be expected either because the goal set with the database was expected to fail, or else because the goal was expected to succeed without finding a value for the variable in question. The same range of bugs may apply in each case, but their effect on program execution in the two cases will be different. This may be exemplified in terms of the 'extra clause' bug. Where failure was expected, this bug might allow the goal to succeed, thus providing the unexpected instantiation. (It will be remembered that only one difference between the ideal and bugged code is allowed, that of the bug itself). Where the goal is expected to succeed without a value for the variable in question, the extra clause must permit a different unification which gives the unexpected value to the variable.

This kind of analysis may be performed for each of the bugs listed above. This implies that under the conditions stated in section 6.3.1, the effects of each listed bug on program execution may be expressed in abstract terms, without reference to the specific code being executed. A summary of the structures which are involved in this expression is given graphically in figure 6. This figure indicates that where a specific symptom is caused by a specific bug, the behaviour caused by that bug can be described in terms of the models of

execution developed in section 6.2. This behaviour must result in either the success or failure of a given goal. A 'template' or 'summary description' can be constructed for each case. This 'template' describes the bug's effect on execution in terms of the models, but without reference to specific pieces of code.

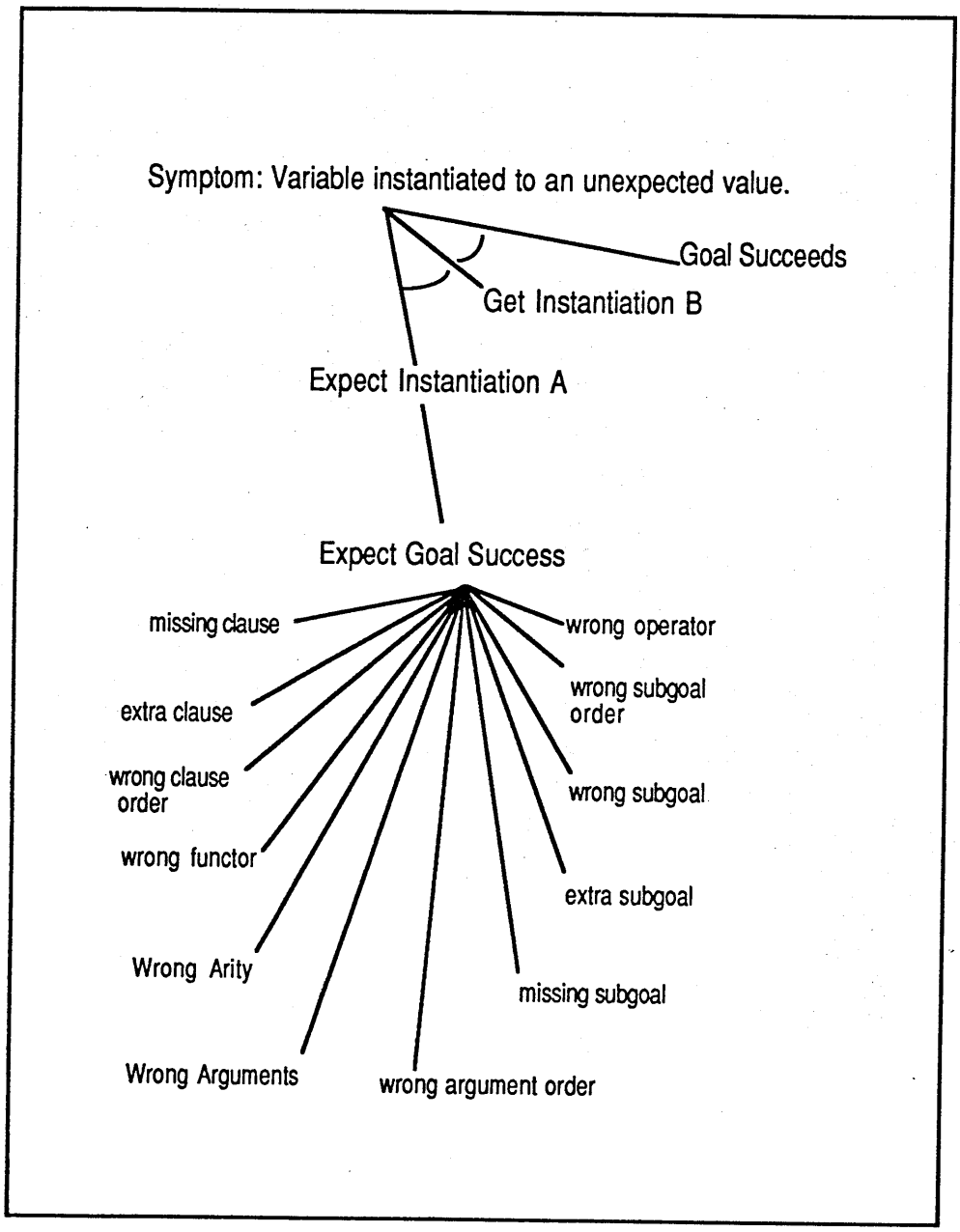
Figure 6. A summary of the relationships of Symptom, Bug, Module, Models of execution, and Templates.



These abstract descriptions or 'templates' may be used to explain the effects of each bug on program execution, or else to check a description of the execution given by a student. If

we assume that the abstract description of a bug's effect can be phrased in terms of the models of Prolog presented in section 6.2, then this will provide a way of formally relating the models to the intended tutoring domain, which is the localisation of bugs in Prolog code. The details of this formalisation are explored in sections 6.4 and 6.5.

Figure 7. The 'Bug Tree' for the symptom 'A variable instantiated to an unexpected value'. (Symptom from Brna et al. 1987).



We indicated in section 6.3.1 that the discussion of this domain would be centred initially around a single symptom, that of 'A variable instantiated to an unexpected value'. The 'bug tree' for this symptom is given in figure 7.

6.3.3 Conclusions to section 6.3.

Section 6.3 refers to Brna et al. (1987) to derive a catalogue of bugs which may be described in relation to the programmer's expectations. The definition of 'Missing', 'Extra' or 'Wrong' modules allows all possible bugs to be defined 'syntactically' in relation to an ideal version of the code. This structure is used in relation to the symptom 'A variable instantiated to an unexpected value' (Brna et al. 1987), to define a range of modules and resultant bugs that the proposed tutoring system will be able to handle. Specific conditions regulating the relationship between the ideal and bugged code are also defined. The symptom, the expectations associated with it, and the range of possible bugs are assembled into 'trees'.

6.4 Formulating abstract descriptions of the effect of each bug.

6.4.1 Introduction.

The discussion so far in this chapter can be summarised as follows: section 6.2 provided models of Prolog execution which, taken together, can be seen as describing a simplified version of Prolog execution; section 6.3 described a simplified, limited and systematised set of bugs which are assumed to operate under given conditions. The model Prolog with the restricted bug catalogue and conditions can be said to provide a much simplified environment which is suitable for novices wishing to learn about debugging strategies.

Having constructed the parts of a simplified domain, it is necessary to define in detail how those parts will interact. Accordingly, in this section, the twelve possible bugs from section 6.3 are discussed in relation to the symptom 'A variable instantiated to an unexpected value'. (See figure 7, and Brna et al. 1987). In general terms, this involves mapping the execution models of section 6.2 onto the bugs of section 6.3 in order to describe how each bug may effect execution. (As this description relates to the issue of 'control flow', it may be seen as a version of the 'Program Misbehaviour Description' of Brna et al. 1987). Once the execution patterns for each bug have been established, then each pattern must be described in terms of the models of section 6.2. These two tasks are undertaken in this section.

6.4.2 Bug effects on Program execution.

In this section the possible execution patterns of each bug are noted.

The discussion in this section assumes that two versions of code are always available, one being the ideal version, the other being the bugged version. If both versions were to be run with the same query, two different execution traces should result. The difference between the trace from the ideal code, and the trace from the bugged code should reveal the effect of the bug, as only one difference between ideal and bugged code is allowed. (See section 6. 3). For each bug discussed in this section, it is assumed that traces are available describing the execution of both ideal and bugged code as a series of numbered statements.

The two execution traces can be compared in terms of the initial query and any resultant subgoals which are generated, eg. a 'failure' in the ideal code trace may occur where the same goal succeeds in the bugged code trace. (The models defined in section 6.2 match a goal with *every* database clause, not just those with the same functor. For a given goal, failure may thus be the outcome in relation to each successive clause in the database). Under the conditions stipulated in section 6.3, and with the requirement that for this

symptom ('A variable instantiated to an unexpected value') both goals must eventually succeed, only certain outcomes are possible, and these can be listed exhaustively. The possible outcomes for one bug, 'Missing Clause' will now be discussed in detail, and then the possible outcomes for all twelve bugs identified in section 6.3.2 will be given in tabular form.

To recap: the symptom under discussion is 'A variable instantiated to an unexpected value'; the bug is 'Missing Clause'; the search strategy does not incorporate backtracking, and only one difference between the ideal and bugged code is allowed. The effect of the missing clause on the execution trace is obviously dependent on the position of that clause in the ideal database. If the missing clause is near the first (top) position in the ideal database, then all the possible unifications which the clauses following it might allow are, as it were, 'moved up' one place. The following pair of databases illustrate this for the goal 'big(X)'.

Ideal Database.

```
big(X):- hungry(X).
hungry(cat).
hungry(iguana).
```

Bugged Database.

```
big(X):- hungry(X).
hungry(iguana).
```

For these databases, the goal 'big(X)' always succeeds, as do all generated subgoals. In each database the execution terminates when the second clause gives 'X' a value. In the bugged database, however, the fact 'hungry(cat)' is missing, allowing the subgoal 'hungry(X)' to succeed on 'hungry(iguana)' thus giving the symptom 'variable instantiated to an unexpected value'. This could only happen in the ideal database if the resolution of 'hungry(X)' and 'hungry(cat)' was forced to fail. The executions which would be shown in the two traces can thus be described by saying that a successful resolution in the 'ideal' trace occurs at the same position as a successful resolution in the 'bugged' trace, but that

these resolutions occur with different clauses. This description can form the basis of a specification which identifies the presence of the bug in question, a 'Missing Clause', if we add to it the stipulation that the ideal database must be one clause longer than the bugged database.

This bug may also show other patterns of execution. A goal in the 'ideal' trace may succeed where a goal in the 'bugged' trace fails if the clause which resolves with the goal in the ideal database is missing in the bugged database. In this case, there must be another clause in the bugged database which gives a successful resolution in order for the variable in the goal to be given a value. (It will be remembered that the symptom here is 'A variable instantiated to an unexpected value'). This situation is illustrated by the following code being run with the goal 'huge(X, Y)':

Ideal Database.	Bugged Database.
big(dog).	big(dog).
huge(red, fish).	unhappy(man):- huge(black, dog).
unhappy(man):- huge(black,dog).	huge(black, dog).
huge(black, dog).	

For the execution with this goal, both the 'ideal' and the 'bugged' trace will show the same result with respect to the first clause, the fact 'big(dog).', as both attempted resolutions will fail to match. The next attempted resolution succeeds in the 'ideal' trace, when the query resolves with 'huge(red, fish).' so that 'X' is instantiated to the value 'red' while 'Y' is instantiated to the value 'fish', and the execution halts. The fact 'huge(red, fish).' is however missing from the bugged database, so that the second attempted resolution shown in the 'bugged' database is with the clause 'unhappy(man):- huge(black, dog).'. This fails, giving a failure in the 'bugged' trace where there is a success in the 'ideal' trace. The execution with the bugged database goes on to resolve the query with 'huge(black, dog).', so that in the 'bugged' trace, 'X' is instantiated to the value 'black', while 'Y' is instantiated to the value 'dog', thus giving the symptom in question.

It is interesting to note that under the conditions specified for this exercise, (see section 6.3), it is not possible to have a situation where a failure in the 'ideal' trace, (with a later success and consequent instantiation of the goal variable) is matched by a success in the 'bugged' trace, if the resulting instantiations are to have different values, ie. if the symptom under discussion is to actually appear. (Under the terms of this discussion, getting the same variable instantiations through different resolutions does not constitute a bug). This is easily explained. If the bug in question is a 'Missing Clause', and a failure in the 'ideal trace' matches a success in the 'bugged' trace, then the clause which causes the failure in the ideal trace must be missing from the bugged database. This would constitute the single allowable difference in the two databases. All subsequent clauses in both databases would thus have to be identical. The first subsequent clause which successfully resolved with the current goal in the ideal trace would also do so in the bugged trace, and the resulting variable instantiation would be the same in both traces, so that the symptom of 'A variable instantiated to an unexpected value' would not be seen. The following code illustrates this point:

Ideal Database.	Bugged Database.
big(dog).	big(dog).
unhappy(man):- huge(black,dog).	huge(red, fish).
huge(red, fish).	huge(black, dog).
huge(black, dog).	

If the query set is again 'huge(X, Y)', then the first successful resolution in the bugged database will be with the second clause, the fact 'huge(red, fish).'. 'X' will thus become instantiated to the value 'red' while 'Y' is instantiated to the value 'fish', and the execution will halt. In the ideal database, the attempted resolution of the query with the second clause 'unhappy(man):- huge(black,dog).' will fail. The traces thus show a failure in the 'ideal' trace where there is a success in the 'bugged' trace. The next attempted resolution in the 'ideal' trace will be the query with the third clause, the fact 'huge(red, fish).'. This will succeed giving the same instantiations as the bugged execution just described. Any

further changes to prevent this happening are illegal, since the absence of the clause 'unhappy(man):- huge(black,dog).' from the bugged database already constituted the single allowable difference between the two databases.

The possible combinations of goal success and goal failure for the symptom 'A variable instantiated to an unexpected value' and the bug 'Missing Clause' (under the conditions defined in section 6.3) may thus be defined as in the entries to Table 1; the other necessary attributes of the bugged database are also listed.

A similar analysis can be carried out for all the twelve bugs listed in section 6.3.2. The possible combinations of bug, goal success, goal failure, and database/trace condition are derived systematically in the same manner and under the same conditions as those given for the bug 'Missing Clause' given above. As these other derivations contribute little of principle to the discussion, they are summarised in Table 2.

Table 1. Possible legal goal outcomes at a given point in the traces of both Ideal and Bugged code for the bug 'Missing Clause' and the symptom 'A variable instantiated to an unexpected value', using models of section 6.2 and database conditions of section 6.3. (Symptom from Brna et al. 1987).

Bug	Ideal Trace	Bug Trace	Bugged Database / Traces
Missing Clause	Success	Success	1 less clause.
Missing Clause	Success	Failure	1 less clause.
Missing Clause	Failure	Success	ILLEGAL

Table 2. Possible legal goal outcomes at a given point in the traces of both Ideal and Bugged code for all bugs of section 6.3. and the symptom 'A variable instantiated to an unexpected value', using the models of section 6.2 and database conditions of section 6.3. (Symptom from Brna et al. 1987).

Bug	Ideal Trace	Bug Trace	Bugged Database / Traces
Missing Clause	Success	Success	1 less clause.
Missing Clause	Success	Failure	1 less clause.
Missing Clause	Failure	Success	ILLEGAL
Extra Clause	Failure	Success	1 more clause
Extra Clause	Success	Success	1 more clause
Extra Clause	Success	Failure	ILLEGAL
Wrong Clause Order	Success	Success	Different clause order. All clauses shared.
Wrong Clause Order	Failure	Success	Different clause order. All clauses shared.
Wrong Clause Order	Success	Failure	ILLEGAL
Wrong Functor	Success	Failure	Functor failure in bugged trace. Dif. functor
Wrong Functor	Failure	Success	Functor failure in *ideal* trace. Different functor.
Wrong Arity	Success	Failure	Arity failure in bugged trace. Different Arity.
Wrong Arity	Failure	Success	Arity failure in *ideal* trace. Different Arity.
Wrong Arguments	Success	Success	Same arity, different arguments
Wrong Arguments	Success	Failure	Same arity, different arguments

Wrong Arguments	Failure	Success	Same arity, different arguments
Wrong Argument Order	Success	Success	Same arguments, different argument order
Wrong Argument Order	Success	Failure	Same arguments, different argument order
Wrong Argument Order	Failure	Success	Same arguments, different argument order
Missing Subgoal	Failure	Success	1 less subgoal; other subgoals of clause same.
Missing Subgoal	Success	Failure	Is failure through lack of subgoal possible?
Extra Subgoal	Success	Failure	1 extra subgoal; other subgoals of clause same.
Extra Subgoal	Failure	Success	Is failure through lack of subgoal possible?
Wrong Subgoal	Success	Failure	Same no. of subgoals; one different.
Wrong Subgoal	Failure	Success	Same no. of subgoals; one different.
Wrong Subgoal	Success	Success	Same no. of subgoals; one different.
Wrong Subgoal Order	Success	Failure	Same subgoals; different subgoal order.
Wrong Subgoal Order	Failure	Success	Same subgoals; different subgoal order.
Wrong Subgoal Order	Success	Success	Same subgoals; different subgoal order.
Wrong Operator	Success	Failure	Different operator
Wrong Operator	Failure	Success	Different operator
Wrong Operator	Success	Success	Different operator

6.4.3 Developing description templates for each bug execution pattern.

Introduction.

This section deals with the detailed mapping of the models of section 6.2 on to the bugs of section 6.3. Section 6.4.2 described the different execution patterns that could be associated with each bug, for a given symptom. A set of conventions is now stated which govern the description of those patterns in terms of the models of section 6.2, and indicates how they can be applied to give a complete description of each pattern. These conventions are also used to develop a taxonomy of bugs, where each bug is classified as either a Search Space, a Search Strategy, or a Resolution bug.

Finally, the description of the execution pattern and the bug classification are combined to give abstract descriptions of the effect and nature of each bug in relation to the symptom 'A variable instantiated to an unexpected value'. These descriptions do not refer to specific pieces of code. They are used to define 'templates' which are intended to support the tutoring and explanation which is required for the simplified domain. The 'templates' are described in more detail below.

Conventions governing the mapping of models onto bugs.

The purpose of this section is to give a statement of the conventions which govern the mapping of the models of section 6.2 onto the bugs of section 6.3, and the reasons for their adoption. Conventions are defined in relation to Resolution, Search Strategy, and Search Space.

These conventions state that bugs shall be classified as follows:

- Any change to functor, arity, argument, or argument order implies a Resolution

bug.

- Any change to clause or subgoal order implies a Search Strategy bug.
- Changes which add or delete clauses or subgoals imply a Search Space bug.

The introduction to this section (6.4.3) mentioned the need for a set of conventions to govern the use of the models of Prolog (section 6.2) to describe the effect of the bugs on program execution. What is required here is a well-structured mapping of the models from section 6.2 onto the bugs of section 6.3. Conventions for this mapping are necessary firstly in order to prevent misunderstandings which might arise from different possible uses of such terms as 'Resolution', and secondly to allow us to associate given types of bugs with the models which facilitate their localisation. This is, after all, the goal of the tutoring. The 'templates' which are structured around this mapping are intended to describe program execution and relate this to the given bugs. They should do so clearly and comprehensibly. The discussion of the conventions leads, later, to a discussion of the 'templates'.

Conventions must first be defined in relation to 'Resolution', since we have to make evaluative judgements concerning the outcome of specific 'Resolutions'..The point is perhaps clarified by saying that a perfectly successful 'Resolution' can constitute a bug, (ie. it succeeds where the 'ideal' trace shows a failure). In this case, no particular aspect of the code such as 'arity' is highlighted as causing a failure, and as the student does not have access to the 'ideal' trace, no specific aspect of the code can be isolated as being 'wrong' in terms of execution. It is the overall *effect* of the execution which constitutes the bug. The Prolog models of section 6.2 do not have the means to describe this, as it involves comparisons between different executions. A vocabulary must thus be developed to permit the expression of such evaluative conclusions.

Confusion may arise from the fact that 'Resolution' can be used in different senses. It can refer to the abstract description of a process which attempts to unify the functors and all the arguments of a goal and one of a number of clauses, thereby possibly binding unbound

variables. Within such an abstract description, alternative 'Resolutions' are possible, so that we might refer to the 'right' or the 'wrong' 'Resolution', depending on the desired outcome.

In a different sense, 'Resolution' can refer to a specific part of a specific execution which attempts to unify a given goal and a given clause in terms of functor, arity, and arguments. This can only be discussed in terms of the success or failure of its individual stages and its outcome. As it is not related to other possible resolutions, it allows no sense of being the 'right' or 'wrong' resolution. It is this latter meaning which is embodied in the models of section 6.2. It is desired that the student should accurately apply these models to describe a specific goal/clause execution and only *then* state that this execution is a manifestation of some specific form of bug. If the goal of tutoring strategies for the localisation of bugs in Prolog is to be well served, then the system must be able to clearly associate the correct bug with one of the models, or a combination of the models, given in section 6.2.

Conventions must also be defined in relation to 'Search Strategy'. 'Search Strategy' is also defined as describing a *local process,* in this case, the one by which clauses are chosen for resolution with a given goal in a given execution. It does not describe the choice of one search strategy in preference to another.

Other conventions must be defined in relation to 'Search Space'. The issue here is slightly more complex. It's sense in the model of section 6.2 is quite clear, relating to the space's de facto ability or inability to prove given goals. If however, we wish to use the term to characterise a certain kind of bug, then the situation is less clear. The way that a bug is labelled is intimately bound up with the action taken to remedy it. Consider the following code, to be executed with the query 'huge(X, Y)'

Ideal Database.	Bugged Database.
big(dog).	big(dog).
huge(black, dog).	huge(red, fish).
unhappy(man):- huge(black,dog).	huge(black, dog).
	unhappy(man):- huge(black, dog).

In both databases the query would succeed in resolving with the second clause, giving X = black and Y = dog in the ideal database trace, but X = red and Y = fish in the bugged database trace. According to the rules of section 6.3, the only allowable difference between the databases is the bug, in this case the extra clause 'huge(red, fish).'. This defines the bug as an 'extra clause' bug. However, if the ideal database is not available for inspection, then the desired result can equally well be obtained by changing the order of the clauses, (ie. by putting 'huge(black, dog) before 'huge(red, fish)'), thus defining the bug as a clause order bug. (For the symptom that we are discussing this is frequently the case, as, by definition, some goal must always succeed, and any other constraints on the code which might make the re-ordering strategy inappropriate have not been articulated).

As another alternative, the functor of 'huge(red, fish).' could be changed to stop the query resolving with it, so that the bug is seen as relating to an aspect of the resolution process. Such clause order and functor changes would violate the conditions defined in section 6.2 by creating two differences between the ideal and bugged code, although this would not be apparent to the student as the ideal code would not be available for inspection.

The re-ordering strategy in this example seems much more closely related to the definition of the Search Strategy model in section 6.2. Questions which concern the presence or absence of a clause seem much more related to the Search Space model. The changing of a functor seems intimately related to Resolution. The question thus arises of how we may structure the mapping of the models of section 6.2 onto the bugs of section 6.3 in an unambiguous and coherent way, so that the changes the chosen bugs imply do not violate the conditions defined in section 6.3. (There is also the more practical question of how the

system is to guide the student onto the target bug and remedy, rather than allowing them to make an alternative analysis implying an action which is 'illegal' in terms of differences between bugged and ideal code).

A possible structure for the mapping is indicated if we consider one final semantic problem. The examples given above involve changing respectively, the number of clauses, the clause order, and one clause's functor. As the Search Space is defined in terms of the collection of clauses which can be used to prove a goal, then each of these three changes will involve changing the Search Space. In fact, any action taken to rectify a bug will involve change to the Search Space, so that in this sense, all bugs are 'Search Space' bugs. This is not very helpful.

Progress can be made by distinguishing the two uses of 'Search Space':

1) A collection of clauses, (the thing to which changes are made).

2) A way of labelling a certain kind of bug.

The first sense is that dealt with in the Search Space model of section 6.2. If we concentrate on the second sense of the term, we must decide what kind of code errors, or corrective changes, we want it to capture. As indicated previously, changes involving clause order seem related to the Search Strategy model, while those concerned with such aspects as functors and arity seem related to the Resolution model. If these categories of changes are excluded, then we are left with those which concern the presence or absence of specific clauses. This seems correct, since it relates the bugs of 'missing clause' and 'extra clause' to the model of the Search Space.

This categorisation can be extended to consider subgoals. The order in which they are considered is dictated by the Search Strategy, so that bugs relating to this order can be seen as Search Strategy bugs. Bugs relating to the presence or absence of subgoals can be seen as Search Space bugs, as can 'wrong subgoal' bugs. Bugs relating to operators do not fit neatly into this categorisation, but can be labelled as Search Space bugs for convenience.

Syntax bugs are not considered here, since it is assumed that these will be caught by the Prolog environment's syntax checker.

In summary, bugs can be named in terms of the changes that are made to rectify them. In the context of the conditions specified in section 6.3, this can be used to structure a mapping from the models of section 6.2 on to the bugs of section 6.3. as follows: Any change to functor, arity, argument, or argument order implies a Resolution bug. Changes to clause or subgoal order imply Search Strategy bugs. Changes which add or delete clauses or subgoals imply a Search Space bug.

A good mapping from the models to the bugs implies that certain bugs should be associated with certain models in a meaningful way. The point of the intended tutoring is that the models should become useful in localising bugs. It is also desirable that, for the sake of consistency, the conditions declared in section 6.3 should be maintained in the analysis which identifies any given bug, (thus, for example, a re-ordering of the clauses would not always be an acceptable solution). The conventions defined above are intended to produce this consistency. It is also intended that they should regulate the labelling of bugs and preclude misconceptions about that labelling.

Classifying bugs in terms of the models.

The conventions established in section 6.4.3 can now be used to label the individual bugs in terms of Resolution, Search Strategy, and Search Space.

The relationships outlined in that section can be shown in tabular form. As the structure the a table will reflect the use to which the entries will be put, this use must be briefly considered first.

The primary purpose of the relationships described in section 6.4.3 is to build 'templates' which describe particular patterns of execution and link them to specific bugs. If we

assume that students have correctly identified the clause that contains the bug, they can be asked to describe the execution of that clause with the relevant goal, and then state the nature of the bug and the model it relates to. We may ask that the execution be described in terms of three of the models from section 6.2, ie. Resolution, Search Strategy, and Search Space.

The first model to be applied will be Resolution, which will be used to describe the attempted resolution of the goal with the head of the relevant clause. If this completes successfully, then Search Strategy will be used to determine whether any subgoals are to be pursued. If the head resolution fails, then the search continues elsewhere. It is at this point that some comment must be made on the search space, as an evaluation of the previously-described execution. If the bugged execution manifests a Resolution or Search Strategy bug, then the Search Space may be described as 'Ok'. If, however, the execution described is seen as manifesting a Search Space bug, (eg. 'extra clause'), then this bug can be given as the description of the Search Space. At this point the student can be asked to state the relevant bug or 'Code Error'. The following table associates Code Errors, the model related to them, and the appropriate comment on the Search Space. The description of the bugged execution in terms of the three models is dealt with more fully below.

Table 3. Each row in the table gives a recognised Code Error, the model associated with it, and the relevant comment on the Search Space. (Code Errors derived from Brna et al. 1987; Models derived from Bundy et al 1985).

Code Error	Model	Search Space Description
Wrong Functor	Resolution	Ok
Wrong Arity	Resolution	Ok
Wrong Argument	Resolution	Ok
Wrong Argument Order	Resolution	Ok
Wrong Clause Order	Search Strategy	Ok
Wrong Subgoal Order	Search Strategy	Ok
Missing Clause	Search Space	Missing Clause
Extra Clause	Search Space	Extra Clause
Wrong Subgoal	Search Space	Wrong Subgoal
Missing Subgoal	Search Space	Missing Subgoal
Extra Subgoal	Search Space	Extra Subgoal
Wrong Operator	Search Space	Wrong Operator

Questions which may be asked of the student.

This section defines the kind of questions that a tutoring system embodying the mappings of models onto bugs may ask of a student.

Given the focus of the domain, the questions will be something like "Where and what is the bug in this program?" or "Where does this trace differ from what you would expect for the trace of a correct solution?". We may assume that at this point the student has access to the following materials:

- The desired result.
- The actual result.
- The bugged trace.
- The bugged code.

What is desired is that the student should indicate the particular clause, clausepart, or aspect of the database which constitutes the bug. For the symptom under discussion, ('A variable instantiated to an unexpected value'), this can sometimes take the form of two quite distinct questions. If a resolution in the 'bugged' trace fails where one in the 'ideal' trace succeeds, then this is the point where a variable value *fails* to be instantiated. (See examples of code above). In order for the execution to produce some value, a later resolution must succeed, giving a point in the trace where an incorrect value *is* instantiated. The question which asks about the location of the bug may thus take the following two forms:

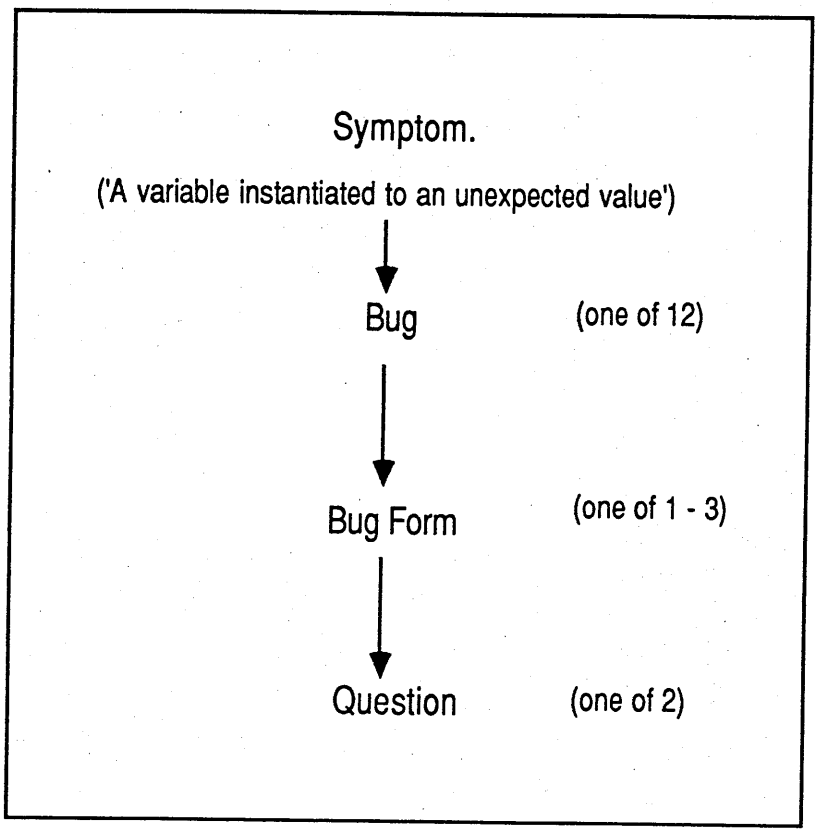
- Where in the trace or code is the 'ideal' value **not** instantiated?
- Where in the trace or code **is** the bugged value instantiated?

Where the 'ideal' and 'bugged' traces both show a successful resolution in the same position, (but with different clauses, see examples above), the two questions actually refer to the same single event, as the 'bugged' trace does not show a failure where the 'ideal' trace shows a success.

These considerations imply that the system must be able to distinguish which question is being asked and must be able to tutor in relation to each one. In terms of an

implementation, this means that where the questions refer to different events, a separate template must be available to describe each event. One will be needed to describe the execution where a variable value *fails* to get instantiated, and another will be required to describe the execution where a 'bugged' value *is* instantiated.

Figure 8. The description template derivation tree for the symptom 'A variable instantiated to an unexpected value' combined with a specific bug, bug manifestation and question. (Symptom from Brna et al. 1987)



Building templates to describe bugged execution.

The appropriate template for describing a specific combination of database, bug, execution, and question can thus be derived from a simple tree structure. As described above, a given

symptom can be caused by one of a catalogue of bugs, each of which may manifest themselves in a small number of ways. In relation to each manifestation, either one or two questions may be asked about the instantiation of the final variable value. This derivation path from Symptom to question is shown graphically in figure 8.

As was stated in section 6.4.1, the purpose of this section is to develop detailed description templates for the various manifestations of each bug described in section 6.3. It is intended that these templates should support the various interactions between student and system that tutoring and explanation in the simplified domain, described in sections 6.2 and 6.3, will require.

Each template constitutes an abstract description of one of the bug manifestations described in section 6.4.2. The terms of the description are those of the models of section 6.2, with such additions as are necessary. The most crucial addition is a set of conventions determining how the terms of the models are to be applied to the ideal and bugged executions. These conventions are stated in section 6.4.3.

Table 3 can be used to build templates which describe the pattern of execution associated with each bug, and the mapping from the Prolog models of section 6.2 onto each bug. This can be illustrated with a pair of databases given previously, which are to be executed with the goal 'huge(X, Y)'.

Ideal Database.	Bugged Database.
big(dog).	big(dog).
huge(black, dog).	huge(red, fish).
unhappy(man):- huge(black,dog).	huge(black, dog).
	unhappy(man):- huge(black, dog).

The bug is an 'extra clause' ('huge(red, fish).'), in the bugged database. If we assume that the student has correctly identified this clause as being the one which manifests the bug,

then we may ask them to carry out the two-part task outlined under the title 'Classifying the bugs in terms of the models' in section 6.4.3.

For the first part they can be asked to describe the execution of the bug-related clause with the relevant goal in terms of the three models from section 6.2. This allows the human or machine tutor to check that they have correctly understood the effect of the bug on program execution. The second part of the task requires the student to identify the 'code error' and the model it relates to in terms of the relationships stated in Table 3.

The first part of the task would follow the pattern described in the paragraph preceding Table 3, and can be expressed here as the following algorithm:

```
For a given goal and clause do the following in order:
1. Check functors for "functors" slot.
        if they match enter 'functors match'
        else enter 'functors fail'
2. Check arity for "arity" slot.
        if functors slot = 'functors fail' enter 'not relevant'
        else check arity
                if arity matches enter 'arity ok'
                else enter 'arity fails'
3. Check arguments for "arguments" slot.
        if arity slot = 'not relevant' or 'arity fails' enter 'not relevant'
        else check arguments
                if all argument pairs unify enter 'arguments ok'
                else enter 'arguments fail'
4. Apply Search Strategy for "Search Strategy" slot
        if arguments slot = 'arguments ok' check subgoals
                if subgoals present and subgoals succeed enter 'subgoals ok'
```

else if subgoals present and subgoals fail enter 'subgoals fail'

else if no subgoals present enter 'success on fact no subgoals'

else enter 'Seek new clause for proof'

Table 4. Alternative slot entries for Functor, Arity, Argument, and Search Strategy slots of templates to describe execution of bugged clause and relevant goal. Rows are not meaningful.

Functor.	Arity.	Arguments.	Search Strategy.
Succeeds.	Succeeds.	Succeeds.	Subgoals: Succeed.
Fails.	Fails.	Fails.	Subgoals: Fail.
	Not relevant.	Not relevant.	No Subgoals: Stops.
			New Clause.

In terms of the code of our example, the goal 'huge(X, Y)' unifies successfully with the fact 'huge(red, fish).'. The template for this version of this bug thus has slots for 'functor', 'arity' and 'arguments' which state that the functor unification, arity matching, and argument unification all succeed. The slot for 'Search Strategy' states that either any subgoals in the clause succeed, or else the clause is a fact so that the search is complete. The alternative descriptors for each slot are given in Table 4.

The second part of the template contains the entries relevant to the second part of the task. These will constitute one of the combinations of Code Error, Model, and Search Space description given in Table 3. In the case of our example code where 'huge(X, Y)' is resolved with the extra clause 'huge(red, fish).', the relevant combination is 'Extra Clause' for the Code Error slot, 'Search Space' for the Model slot, and 'Extra Clause' for the Search Space description slot. The complete template is given in Table 5.

Table 5. Complete template to describe the execution and analysis of version 1 of the bug 'Extra Clause' when the relevant clause has no subgoals. 'Version 1' implies that the traces of both bugged and ideal databases show a successful resolution at the same point.

Template Slot.	Template Entry.
Functor.	Succeeds.
Arity.	Succeeds.
Arguments.	Succeeds.
Search Strategy.	No Subgoals: Stops.
Search Space.	Extra Clause.
Code Error	Extra Clause
Model	Search Space.

Template substitution.

Earlier it was pointed out that when the 'bugged' trace shows a failure on a clause where the 'ideal' database shows a successful resolution, then two distinct questions about the location and nature of the bug could be asked, (in the case of the symptom 'A variable instantiated to an unexpected value'). One question asks where the expected value *fails* to get instantiated, while the other asks about where the unexpected value *is* instantiated. The issue of providing templates to support both forms of question about the bug is simply resolved. The failure can be described by a template specific to that version of the bug. The second question implies an answer which describes a successful resolution in the bugged database (occurring after the failure). In most cases, a successful resolution for a specific bug will already have been described in a template, albeit in relation to a different manifestation of the bug. Once the correct point in the trace relating to the second question has been identified, all that is required is that a 'success' template relating to a different

manifestation of the bug be used to describe it. This can be illustrated with the following code, with the goal 'huge(X, Y)'.

Ideal Database.	Bugged Database.
big(mad, dog).	big(mad, dog).
huge(red, fish).	big(red, fish).
huge(black, dog).	huge(black, dog).

A 'wrong functor' in the bugged database leads to a resolution failure on the second clause, ('big(red, fish).') where the ideal database execution succeeds. The ideal database thus returns X = red, Y = fish, while the bugged database returns X = black, Y = dog, because the bugged database execution succeeds on the third clause, 'huge(black, dog).'. A different manifestation of the same bug is however possible, where the bug occurs earlier. Consider this code, to be executed with the same goal:

Ideal Database.	Bugged Database.
big(mad, dog).	huge(mad, dog).
huge(red, fish).	huge(red, fish).
huge(black, dog).	huge(black, dog).

The ideal database execution is unchanged, but the bugged database execution now succeeds on the first clause, 'huge(mad, dog).' giving X = mad, Y = dog. In this case, the first and second form of question about the bug would both refer to the same successful resolution. The template used to describe this would contain the same entries as the template used to describe the second, successful, resolution in the first bugged database that was considered, and can be used to answer the second form of question relating to that manifestation of that bug.

6.4.4 Conclusions to section 6.4.

This section takes the subset of Prolog defined in section 6.2 and the restricted bug catalogue and conditions defined in section 6.3 and regards them as a simplified

environment which is suitable for novices wishing to learn about debugging strategies. The section then sets out to describe the detailed structure of that domain.

The possible effects of each bug on program execution are described in section 6.4.2. A set of conventions is established to govern the process of describing these effects in terms of the models of section 6.2. These conventions are also used to define a taxonomy of bugs. This taxonomy, along with the distinct database features of each bug and the description of its patterns of execution are used to develop a detailed templates in relation to each bug. The purpose of the templates is to support the tutoring and explanation which is required in the simplified domain. These templates give an abstract account of the possible executions of each bug, and do not refer to individual pieces of code.

The form of the templates is dependent on their intended use in specific student-system dialogues. The dialogues involve two tasks. The first involves the description of a specific resolution attempt in terms of the Resolution and Search Strategy models. An algorithm for this task is given. The second task involves the evaluation of this execution in terms of statements describing the Search Space and the perceived Code Error. The terms for describing the particular execution are listed, as are the possible combinations of Code Error, model, and Search Space description given the stated conventions linking the models to the bugs. An example of a template is also given. The questions which may be asked in relation to each template are considered.

6.5 Re-formulating the Models of Prolog for tutorial dialogues.

6.5.1 The dialogues required.

The purpose of developing the definition of the domain given in this chapter is that a tutoring system should be built which embodies the definition. Sections 6.2, 6.3, and 6.4 have defined, respectively, simplified models of Prolog, a simplified catalogue of bugs,

and a set of abstractions, (the templates), which map the models onto the bugs and describe their effect upon program execution. The tutoring of this domain will necessarily involve interactions between the student and the proposed system. The final interaction desired is that the student should successfully produce the descriptions contained in the templates given under the title 'Building templates to describe bugged execution' of section 6.4.3, when they are presented with a set of code and results which manifest the relevant bug.

This desired interaction has many prerequisites, among which are:

1) The student must understand the models of Prolog developed in section 6.2 and be able to apply them accurately.

2) The student must be able to accurately identify the bugged clause or resolution.

It is the first prerequisite which concerns us here, as it may require that the individual viewpoints be tutored directly, or that the student's use of them be critiqued by the system. The accurate understanding and application of the Prolog models by the student is intended, very generally, to arise from two main forms of interaction between student and system. These are Tutoring and Explanation. The issue of whether these are in principle separate activities is not pursued here.

The point to be made is that, in implementational terms, these activities make separate demands upon the system. Any tutoring on the nature and application of the Prolog models requires that the student be able to input to the system statements which describe specific Prolog executions in terms of the models. These statements are then processed by the system and a (hopefully) suitable reaction computed. The initial flow of information is thus from student to system. Explanation is here seen as a description of execution flowing from system to student. Both these forms of description will require a language suited to the task.

The models as given in section 6.2 are not well-suited to active tutoring and explanation since (with the exception of some parts of the Resolution model), they are largely

declarative in nature. This statement is intended to mean that while they describe models of execution in abstract terms, they do not indicate which part of which model is applicable at any particular point in a specific execution. They thus do not help the students to *apply* their knowledge of the models, and do not make it easy for the system to justify the selection of a particular model or modelpart as being relevant to a particular stage of execution.

The student's application of their knowledge can be at two levels. The first relates to a simple description of execution, and involves selecting one part of one of the models as being appropriate to describe the next step in the execution. The second is more concerned with localising bugs in the code. This requires that the student should choose a particular model, and by implication a particular range of possible bugs, as being most relevant to a given bugged execution. Putting this another way, the student should learn how to apply the different models to localise different bugs. This is the skill that the proposed system is intended to tutor.

What is required to describe execution is a set of 'if ... then' or 'production' rules which embody the models, and which specify the conditions of application for each model part. In other words the models must be 'proceduralised'. In this form the models would provide the student with a means of choosing which part of which model to apply, of predicting the effects of that application, and of justifying their decision. Such a rule-set would also provide the system with a means of structuring and justifying explanations of execution.

6.5.2 'Procedural' versions of the Prolog models.

The models of section 6.2 were developed from the work of Bundy et al. (1985). The models given in this section add 'procedural' information to those of section 6.2, which states the execution conditions in which each part of each model should be applied. The

motivation for this exercise is given in section 6.5.1. The 'Database' model, being largely concerned with syntax, is not seen as relevant to this exercise.

The procedural form of the Search Space Model.

SSP1) When trying to prove a goal and generated subgoals, the same initial Search Space must always be used.

SSP2) When all search Space clauses have been tried without success, a goal fails.

The procedural form of the Search Strategy model.

SST1) If there is a goal, try to resolve it with the head of the first/next database item.

SST2) If the head resolves, consider subgoals.

SST3) If there are untried subgoals, set the first as a goal with the full Search Space.

SST4) If there are no subgoals, or none left untried, the goal which has resolved with the head of the clause succeeds.

SST5) If the resolution of a goal and clausehead fails, or if a subgoal of the clause fails, the whole resolution fails.

SST6) If a resolution fails, try to resolve the goal with the next item in the database.

The procedural form of the Resolution Model.

R1) Check the functors. If they unify, continue.

R2) If the functors do not unify, fail.

R3) If the functors unify, check the arity.

R4) If the arity is different, fail.

R5) If the arity is the same attempt to unify the arguments.

R6) Take the first untried argument from the goal and the first untried argument from the clause. They unify if they are one of the following:

- identical constants.
- a variable bound to the value of the other, which is a constant.
- an unbound variable and a constant.

- a pair of unbound variables.
- two variables bound to the same value.

R7) If any pair of arguments do not unify, fail.

R8) If the functors have unified, and the arity is the same, and there are no arguments or none left untried, the resolution of goal and clausehead succeeds.

R9) If a goal contains an operator, and can be evaluated to 'true' with the current variable bindings, the goal succeeds.

R10) If a goal contains an operator, and cannot be evaluated to 'true' with the current variable bindings, the goal fails.

These individual model parts are also used in an abbreviated form. They are shortened so that they can be offered as menu items, thus forming a description language that can be used by the system to take input from the student.

Although backtracking is not modelled in these models, some care has been taken to formulate them in such a way that they could be extended to describe backtracking in the future. Items SSP2 and SST6, for instance, could be extended to describe the more complex behaviour that failure initiates when backtracking is included in the description.

6.6 Conclusions to Chapter 6.

Section 6.2 defines 4 models of Prolog execution derived from the four proposed by Bundy et al. (1985); (three of these are to be used with the tutoring system). Some parts of their models, such as backtracking and the use of the cut, are omitted. Other parts, such as the generation and processing of subgoals, are changed. It is suggested that they could form the starting point for a series of more complex models which could be used for tutoring at a more advanced level than is here intended.

Section 6.3 assembles a catalogue of bugs which the intended tutoring system will be built to handle. These are defined through the work of Brna et al. (1987). An analysis in terms

of 'Missing', 'Extra' or 'Wrong' modules allows all possible bugs to be defined 'syntactically' in relation to an ideal version of the code. When used in relation to the symptom 'A variable instantiated to an unexpected value' (Brna et al. 1987), the analysis allows a range of modules and resultant bugs to be defined for the proposed tutoring system. Specific conditions are stated to regulate the relationship between the ideal and bugged code. The symptom and bugs are expressed graphically as 'trees'.

Section 6.4 combines the two simplified domains of sections 6.2 and 6.3 to provide a mapping from the models to the bugs. The different possible patterns of execution that each bug can show are noted, along with the distinct database features relating to each one. A set of conventions are established which allow the execution patterns for each bug to be described in terms of the models of section 6.2. These conventions are also used to define a taxonomy of bugs.

The descriptions of the execution patterns and the taxonomy are used to develop detailed templates which describe each bug and its effect on execution in relation to the symptom 'A variable instantiated to an unexpected value' (Brna et al. 1987). These templates are intended to support the tutoring and explanation which is required in the simplified domain. The templates also take account of the fact that for some patterns of execution, two questions can be asked about the precise nature of the bug.

Section 6.5 re-formulates the Prolog models of section 6.2 as a series of procedural statements in order to facilitate dialogue between system and student. Each statement gives the conditions under which the relevant part of the model should be applied.

As stated in chapter 2, the intended research direction is to tutor the application of the models of Prolog in localising bugs rather than to build a 'debugger'. The use of the 'Modules', 'Symptoms' and 'Code Errors' (Brna et al. 1987), in conjunction with the additional stipulations and structures detailed above, makes it possible to build a closed and structured 'world' which only requires a fairly simple 'black box' for a 'bugfinder'. Our

efforts, as demonstrated in the body of this chapter, can then be concentrated on developing structures and mechanisms which describe the various bugs and their effects upon execution in terms of the models we wish to use. Other mechanisms can then be developed to tutor this skill. The emphasis is thus not on building a system which can simply find the bugs, but on building one which can serve our pedagogical goal.

These state that the student should develop the ability to view Prolog execution in terms of the models defined in sections 6.2 and 6.5, and that they should use these models to localise bugs. The use of an ideal version of the code allows us to concentrate on promoting these goals, with the system setting the agenda and determining which specific bugs may be dealt with in a given tutoring episode.

The structures detailed in this chapter can be used to implement the knowledge representations of a tutoring system.

The core tutorial dialogues will involve:

1) asking the student to describe the execution of Prolog code;

2) asking the student to identify the bugged clause;

3) asking the student to detail the effects of the bug on program execution, and to classify the bug.

Chapter 7. VIPER: Viewpoint-based Instruction for Prolog Error Recognition.

7.1 Introduction.

This chapter describes, in outline, the implementation of a tutoring system, (VIPER), which is designed to exploit the conceptual structures detailed in chapter 6. One part of these structures describes three sets of procedural rules, each of which constitutes a partial model of Prolog execution. The other parts describe a set of bugs which VIPER is designed to recognise, and a mapping from the execution models onto the bugs. A description of the implementation is considered to be necessary due to the novel use of multiple viewpoints in VIPER, and the claims that the author wishes to make regarding the system architecture as a design solution, and possible extensions to this architecture. The most basic claim is that, while the system described here is only adaptive in limited ways, it provides mechanisms with which highly adaptive tutoring could be accomplished, using well-researched methods.

The operational goals of VIPER are as follows: The student is to be presented with novice-level combinations of code and query which contain a single bug. This bug constitutes the only difference between the bugged code and an ideal code. The student's task is to locate and identify the bug, and describe its effect on the execution, using the three procedural models given in section 6.5.

The pedagogical goal of VIPER is that the student should develop the ability to view execution in terms of these models so as to localise possible bugs.

Before describing VIPER in detail, some introductory remarks are necessary. The first topic to be addressed by these relates the absence of backtracking in VIPER to the notion of upwardly compatible models. As noted in section 6.2.5 backtracking was excluded in the prototype version of the system as our intention was to use, in the first instance, examples

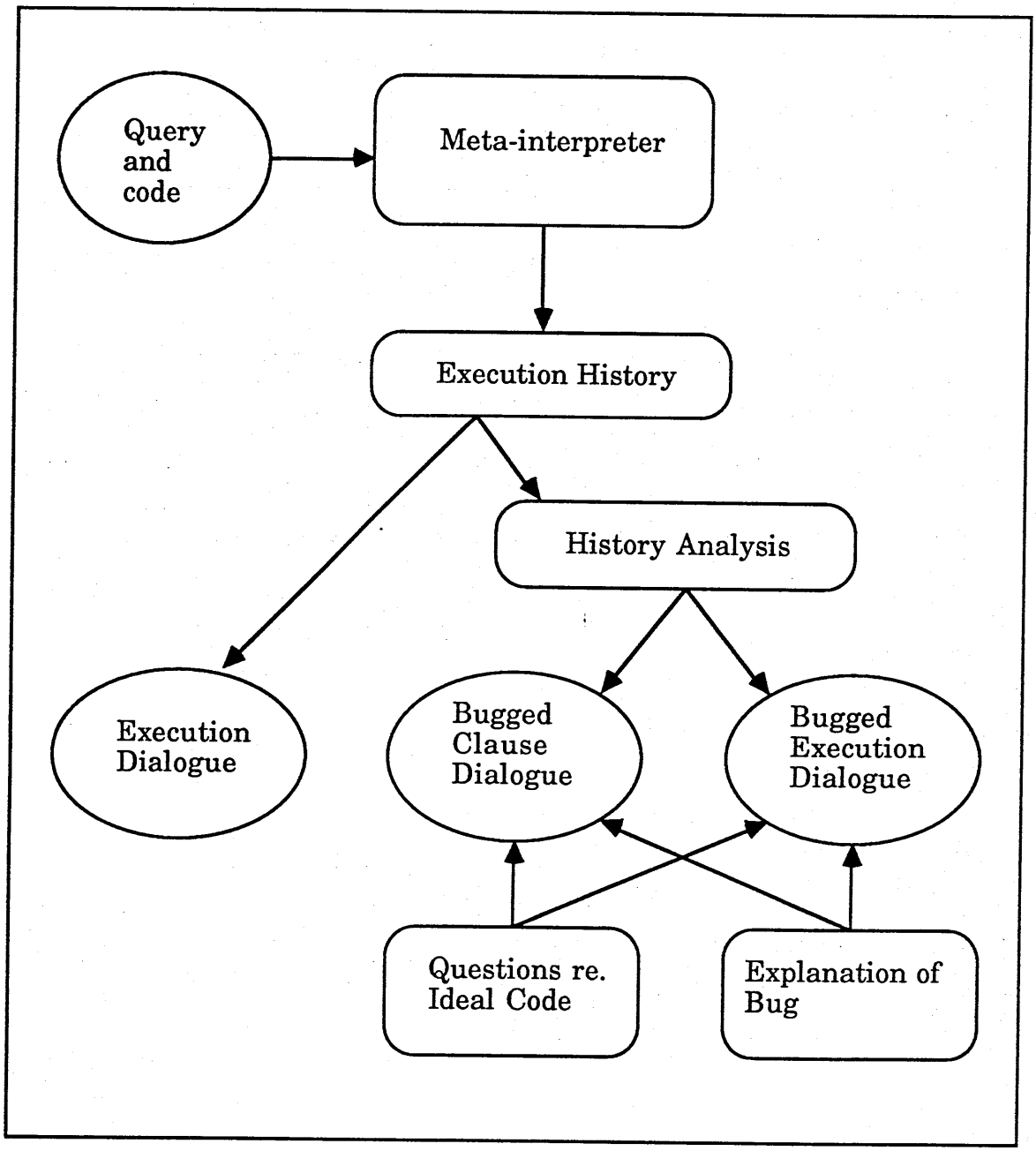
which did not depend on backtracking to produce their results. It was envisaged that a hierarchy of models could be defined which described Prolog execution at increasing levels of complexity until the full range of Prolog behaviour had been covered. It was assumed that, given sufficient time, the prototype system could be expanded to cater for all of these models and their combination with operators and heuristics to form viewpoints.

'Upward compatibility' was discussed in sections 2.1.3 and 6.2.5 in relation to QUEST, (White and Frederiksen 1986). White and Fredriksen describe a series of increasingly-sophisticated models which are intended to represent increasing levels of knowledge and skill in the analysis of electronic circuits. A crucial connection between the models is 'upward compatibility', which ensures that a simpler model can be extended or refined so as to match the next more complex one with a minimum of effort and reconceptualisation. Section 6.2.5 indicates how the Search Strategy and Resolution models could be extended to include a description of backtracking, and could thus form the core of viewpoints which deal with the full range of Prolog behaviour.

Each part of VIPER will be considered in terms of its purpose, its structure, and, where relevant, its inputs and outputs. Since VIPER was itself implemented in a variety of Prolog, (LPA MacPROLOG 3.0), this description will sometimes need to use the vocabulary associated with that language. All code was implemented by the author. A schematic outline of the system is given in figure 9. This shows a meta-interpreter which, when run with a query and suitable code, produces an execution history in terms of the three procedural models of section 6.5.

This history can either be used as it stands to facilitate a dialogue with the student which simply describes the execution that took place, or, if execution histories for both ideal and bugged code are available, they can be analysed to determine the nature of the bug and its effect on execution.

Figure 9. The structure of VIPER.



This analysis enables two other dialogues to take place, one to determine which clause is bugged, the other to describe the effects of the bug on the execution of the code. During these last two dialogues, a menu allows questions to be asked about the ideal code, and an explanation facility allows a complete explanation of the current bug to be given. The questions which may be asked (via the "Questions" menu) are based on the "access"

operators of chapter 5 which may be used to retrieve information in relation to each viewpoint. The explanations which are also provided via the menu are structured around the "inference operator" of chapter 5, which makes explicit information which is only implicit in the models.

Viper was not developed to the point where it could use the third type of operator described in chapter 5, (the operator which introduced new information related to the system state), or the heuristics which state the area of application for each viewpoint. The third class of operator was omitted in order to keep the implementation project within the relevant constraints of time and space. We believe that with further development this class of operator could be used to modify the models used by VIPER so as to represent student misconceptions or errors. In relation to the heuristics which state the area of application for each viewpoint we would claim that these are implicit in the very domain formulation on which VIPER's design is based, (ie. the formulation given in chapter 6). In this formulation, and in the system-student dialogues which are described in this chapter, each viewpoint is exclusively related to the localisation of specific classes of bug. VIPER thus represents a partial implementation of the structure for viewpoints which is given in chapter 3.

This chapter describes in turn, the meta-interpreter itself, the recording and analysis of the execution histories it produces, and the three forms of dialogue which the recorded histories support.

7.2 The Meta-interpreter.

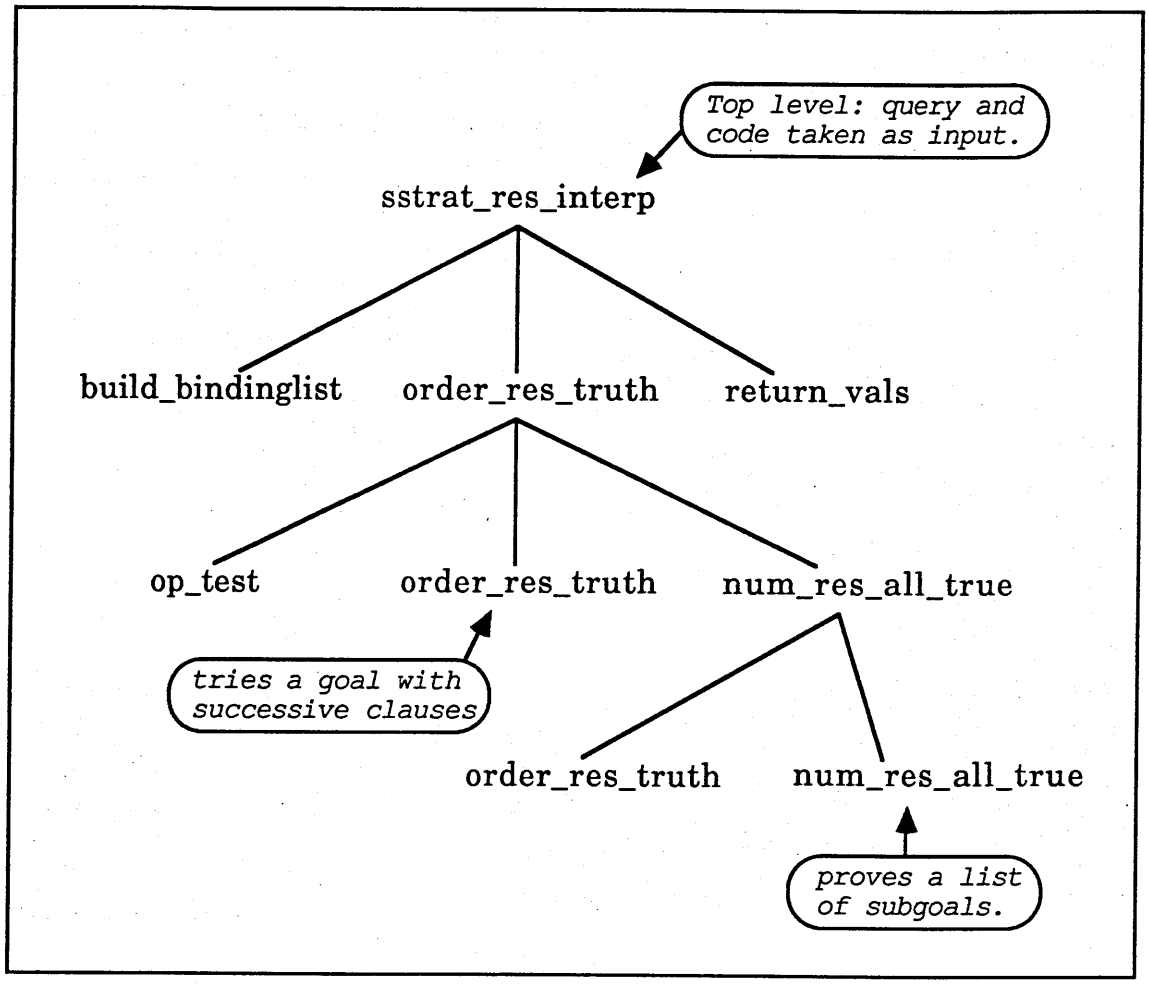
7.2.1 The meta-interpreter: top level.

This module is intended to allow the execution of an input goal and database to be 'watched' as it proceeds through its various stages. More correctly, the meta-interpreter creates a detailed history of the execution, and it is the navigation through this history that a system user will 'watch'. The meta-interpreter is built to reflect the structure of the three (partial) models of Prolog execution described in chapter 6. This means that it describes the execution in terms of Resolution, Search Strategy, and Search Space. In this implementation the execution does not include backtracking, and thus does not require an interpretation of the 'cut'. At its current level of development it also excludes embedded variables, 'true', 'not', and 'false' statements, and some of the operators normally associated with a Prolog environment. The execution history is stored in facts which are asserted as the execution progresses. A single execution will thus produce a complete set of facts relating to all three of the Prolog models specified in chapter 6. Provision is made for alternative meta-interpreters (eg. one using a random searching strategy), to be used as part of VIPER, if this is thought to be tutorially desirable. (For example, the purpose and function of the true search strategy of Prolog could be demonstrated by comparing the execution history it produces with that of a meta-interpreter utilising a random searching strategy).

The top-level call to the meta-interpreter defines the kind of meta-interpretation that is required, (ie. whether it should exhibit a correct or incorrect subset of Prolog behaviour). The most version of VIPER described here uses the correct subset. For this a top-level predicate provides a bindinglist to hold any values bound to variables in the input query, and asserts an execution history fact to describe the start of the execution. Other subgoals of this predicate initiate the execution of the query with the database, and retrieve any values which are instantiated in the bindinglist as a result of it: (see 'sstrat_res_interp' in

figure 10). A second definition of this predicate caters for the failure of the input query by returning 'no' in place of a bindinglist.

Figure 10. A partial call graph for the meta-interpreter: 'order_res_truth' checks a goal against successive database clauses; 'num_res_all_true' proves a list of subgoals.



The inputs to the meta-interpreter are a query and a database, both in the form of lists. The clauses of the database list are numbered in sequence from 1 to *n*, *n* being the length of the list. Its outputs are the result of the execution, which may be a 'no', or a bindinglist of values for any variables in the initial query. Also output are the history facts which record the execution of the query and database.

7.2.2 Implementing the Resolution model.

This module of the meta-interpreter describes the resolution of a single goal and clausehead or fact. It checks that first the functor,and then the arity of the input goal and clause are the same, and fails the resolution if they are not. At each stage, a fact recording success or failure is asserted. If functor and arity match, the module attempts to unify each pair of arguments in turn, failing the resolution if any pair cannot be unified. History facts are asserted for each pair. At the conclusion of this 'head' resolution, another fact is asserted to record the result. The predicate handling this 'head' resolution, ('do_resolve'), is called at appropriate times by the code which defines the search strategy. The setting of subgoals as new goal literals is a separate issue which is also coded as an aspect of the search strategy.

The head resolution predicate, ('do_resolve'), takes as input a goal and clause in list form, and a bindinglist which contains values for any variables in the goal and the clause. It delivers output in the form of a bindinglist which is updated as necessary to reflect any new bindings made in the head resolution.

The unification of successive pairs of arguments, and the entry of any new bindings in the bindinglist, are carried out by a predicate which attempts to unify the heads of any two lists of arguments it is given, and calls itself to recurse if it succeeds. If either of the heads is a variable, then the input bindinglist is accessed to retrieve any value entered there for that variable. Any new bindings which result from the unification of a specific argument pair are entered into the bindinglist. The outcome for each pair of arguments is asserted as a history fact.

Definitions of the predicate are required for arguments which are:

- two identical constants;
- a constant and a variable;

- two variables;
- two arguments which cannot unify.

The procedure which carries out the argument checking, ('check_args'), takes as input the lists of arguments from the goal and clause, and an input bindinglist. It gives execution history facts and an updated bindinglist as output.

The predicate called to access the variable values stored in the bindinglist, ('bind'), needs to make a different set of discriminations. The different possible argument combinations require different actions to be taken.

The different combinations are here followed by the actions appropriate to each one:

- a bound variable with a constant; check the variable value.
- an unbound variable with a constant; bind variable to constant value.
- two unbound variables; bind together.
- two variables bound together, but with no value; no action.
- a bound variable with an unbound variable; bind together.
- two bound variables; check values the same, otherwise fail.

'Bind' calls a number of predicates to search and edit the bindinglist which has the form of a list of lists. Its inputs are the two arguments to be unified, and a bindinglist. The output is an updated bindinglist.

7.2.3 Implementing the Search Strategy model.

The purpose of the model is to allow explicit reports of the different stages of the search to be made.

Reports are made of the following events:

- starting to resolve the current goal with the first/next database item.

- current resolution has failed, so quit it.
- current goal successfully resolves with a fact.
- current goal successfully resolves with a clausehead.
- current goal successfully resolves with a clausehead, and the subgoals succeed.
- a subgoal succeeds.
- all subgoals succeed.
- a subgoal fails.

These reports are made by a predicate, ('order_res_truth'), which attempts to prove the goal that is input to it along with the database. This goal is checked to see whether or not it contains an operator. (A range of thirteen infix operators are accepted by the meta-interpreter). Where an operator is present, the goal is evaluated using the relevant bindings from the input bindinglist. Where the goal contains no operator, 'do_resolve' is called to attempt the unification of the goal with the first item on the database that is also input. If that resolution succeeds, and the item is a fact, 'order_res_truth' asserts an execution history fact to that effect, and terminates. Where there are subgoals to be proved, another predicate ('num_res_all_true') is called, and the resulting bindinglist given as the output of the predicate. Should the resolution fail, 'order_res_truth' calls itself to recurse with the tail of the list which forms the input database. Where that tail is an empty list, the end of the search space has been reached, and the current goal must fail, as backtracking is not incorporated in this implementation. A fact recording an empty database is asserted, and the call is made to fail.

The predicate 'order_res_truth' thus requires definitions for the following cases:

- the input database is an empty list.
- the input goal contains an operator but no variables, and succeeds.
- the input goal contains an operator but no variables, and fails.
- the input goal contains an operator and variables, and is equivalent to a truth condition.

- the input goal contains an operator and variables, and requires that the variable binding list be updated.
- the input goal resolves with a fact in the database.
- the input goal resolves with the head of a clause in the database.
- the input goal fails with previous predicates so the next database clause must be tried.

The need to evaluate goals which contain operators means that other facts relating to these must be asserted by 'order_res_truth', although these facts are subsequently interpreted by the user and VIPER as relating to the Resolution model. (These aspects of execution are also discussed in chapter 6.4.2 as a part of the Resolution model).

'Order_res_truth' thus needs to assert execution history facts for the following cases:

- the evaluation of a goal containing an operator begins.
- the evaluation of a goal containing an operator succeeds.
- the evaluation of a goal containing an operator fails.
- the evaluation of a goal containing an operator has failed, and thus the current attempt at resolution has failed.

This last case may not be seen as strictly necessary, but is included to maintain consistency with actions taken when other attempts at resolution have failed. The actual evaluation of the goal is undertaken by another set of predicates which are detailed below.

In the case where 'order_res_truth' is called with an empty database, a fact must be asserted which is subsequently interpreted as relating to the Search Space model. This is consistent with the procedural formulation of the models which is given in section 6.5.

The inputs to 'order_res_truth' are a goal and a full database, a list representing the database elements yet to be searched in an attempt to prove the goal, and a list of the bindings for each variable in the goal and the clause. Its outputs are the asserted facts, and,

if the call succeeds, an updated bindinglist.

It was stated above that 'order_res_truth' calls 'num_res_all_true' to attempt the proof of any subgoals which become relevant as a result of resolving a goal with the head of a clause in the database. This is consistent with the formulation of the models given in section 6.5. The setting of subgoals as new goal literals is viewed as an aspect of search strategy, since the goals have to be proved in a given order. For each subgoal in turn, 'num_res_all_true' builds a bindinglist for the variables in the subgoal, and calls 'order_res_truth' with the full database of clauses available at the beginning of the meta-interpreter execution. This initial bindinglist for a subgoal must take account of any bindings made in the head resolution of the clause. This is achieved by giving the bindinglist which results from the head resolution as input to 'num_res_all_true'. If a subgoal succeeds, 'num_res_all_true' updates its input bindinglist and recurses, with the resultant list and the list of remaining subgoals as input to the recursion. It also keeps a count of how many subgoals have been proved, and uses this count to assert facts relating to the success or failure of each subgoal. If called with an empty subgoallist, the input bindinglist is given as the outputlist, and a fact recording the success of all subgoals is asserted. If a goal fails, the parent goal is also failed.

This indicates that 'num_res_all_true' should be defined or the following cases:

- the subgoallist is empty as all subgoals have succeeded.
- a new subgoal is to be proved.
- a subgoal fails.

The inputs to 'num_res_all_true' are thus a list of subgoals to be proved, the full database, the current bindinglist, and the number of the current subgoal. Its outputs are the asserted facts, and, if it succeeds, an updated bindinglist.

Frequent mention is made in the preceding paragraphs of the bindinglists used to store the values associated with each variable. These lists are built by first creating a list of all the

variables in the input goal, and then creating a place in the binding list for each of the listed variables. A variable is not referred to by its name, but by the unique identifier that the host Prolog environment assigns to it. For each variable, a search is carried out to ensure that it is not already present in the bindinglist. The following is an example of a bindinglist for three distinct variable bindings:

```
[[rabbit, _167], [white, _168, _142], [_893, _241, _267, _765]]
```

Each element of the bindinglist is a sub-list, the head of which represents the value to which the variable or variables in the tail are bound; (see the first and second elements in the example bindinglist above). Uninstantiated variables can thus be bound together by inserting both variable identifiers into the same sub-list tail; (see the third element in the example bindinglist above) New elements in the bindinglist are created for any variables which are present in the resolving clause, but not in the goal.

When the proof of a given goal is complete, the resultant bindinglist is searched to find the values associated with each of the variables in the goal. These values are assembled with the variable identifiers into a bindinglist which is given as the proving predicate's output.

The last predicate which requires description in this section deals with the evaluation of goals which contain operators. As stated above, the meta-interpreter can support a sub-class of thirteen of the infix operators which are normally associated with Prolog. the 'op_test' predicate tests for the presence of an operator by converting the goal into a list via the "univ" operator, and then examining the head of that list. A goal with an operator but no variables is simply set as a goal in the host Prolog environment. Other goals containing operators are classified as either 'truth-conditional', (ie. they are either true or false and their success would not require any updating of the bindinglist), or 'evaluative', (ie. their success would require that a new value be entered for a variable in the binding list). The truth-conditional goals are proved (or not) in the host Prolog environment. To do this, a procedure which unifies each of the variable identifiers from the bindinglist with the

relevant value, and then sets the input goal to the host environment, is preceded with a call of "not not". This tests the truth or otherwise of the goal without binding the variable identifiers in the host environment to the values held in the meta-interpreter's bindinglist. (If the actual goal succeeds, the variables bound before calling it are unbound by the failure of the second 'not').

Goals which contain operators and which would bind a variable to a new value are tested by identifying dummy variable names with the relevant values and setting the goal to the host environment to obtain the outcome. This is then entered into the bindinglist. Where variables are bound, execution history facts are asserted with the relevant information.

7.2.4 Implementing the Search Space model.

This model is largely implicit in the structures described above. The assertion of execution history facts relating to the search space is only required to describe three conditions.

These are:

- when a top level call to the meta-interpreter initiates the proof of an initial query with a given database;
- when a subgoal is set as a new goal literal with the complete database;
- when the searching predicate is called with an empty database.

For the first of these cases, the fact is asserted by the call to the top-level predicate which defines the type of meta-interpretation that is required. The second case is dealt with by calls to 'num_res_all_true', the predicate which attempts to prove a list of subgoals. The final case is dealt with by a call to 'order_res_truth'.

7.3 Recording and analysing the execution history.

7.3.1 Asserting the execution history facts.

The execution of a query and a database is recorded as a set of asserted facts. The arguments of these facts contain data relating to the nature of the code, (ie. bugged or ideal), the number of a particular step in the total execution, and the nature of the action being recorded. Where relevant, the goal, the number of the resolving clause and the clause itself are also recorded.

The predicates which assert these facts obtain the data from a number of sources. A fact indicating whether the current execution is of ideal or bugged code is asserted before each execution is begun. For each execution history fact that is asserted, this information is retrieved and included in the arguments. A count is kept of execution steps so far, and augmented for each execution history fact assertion. The nature of the action being recorded is given to the fact assertion predicate as input by the predicate making the call. This input takes the form of one of a range of symbols, and is an adaptation of the method used by Eisenstadt (1984, 1985), which is described in section 2.3.2. Each symbol used is related to one or more of the model-parts given in section 6.5. The goal, clause number and clause are similarly given as input to the asserting predicate.

Four different predicates are required in order to assert the facts relating to different aspects of the execution. A general predicate, ('trace'), deals with all assertions except those which relate to subgoals, variables, and operators. It takes as input a symbol indicating a specific action, the relevant goal, and the relevant clause. The goal and clause are converted into their 'printing' form, (ie. into atoms), and included in the asserted fact. An example set of history facts for a simple execution is given in appendix 1.

A separate predicate records the setting of subgoals as new goal literals, and their eventual

success or failure. This takes as input an execution action symbol, and a number indicating the order of the subgoal in the list that have to be proved. This data does not have to be converted to an atomic form.

A separate predicate is required to record the binding of variables, due to the characteristics of the host Prolog environment being used. These characteristics mean that the variable identifiers used by VIPER cannot be asserted alone in facts to record the execution history, even when they have been converted into atoms. (The host environment still reads them as identifiers and will update the asserted fact to reflect any subsequent binding of the variable in question). The variable identifiers must thus be further converted from atoms into strings of characters, and asserted in the facts in this form. The inputs to this predicate are an action symbol, and the two terms which are or are not unified.

A final predicate is required to record the histories of goals containing operators. As the goal is not resolved with a clause, only two arguments are given as input, the action symbol and the goal itself. The goal is converted into an atom, and the fact asserted.

An example trace of asserted execution history facts is given in appendix APPENDIX1.

7.3.2 Analysing the execution history facts.

The bug-description structures developed in chapter 6 assume that the execution of an ideal database is to be compared with that of a bugged database. The sets of facts produced by each execution can thus be compared to determine firstly where they differ, and secondly what the cause of that difference could be; (ie. what bug is present in the bugged database). This analysis is made relatively simple by the conditions stipulated in chapter 6, that there shall be only one difference between the ideal and bugged code, and that this difference should constitute one of the range of allowed bugs.

The point where the two execution histories diverge is determined by comparing the facts

for the same-numbered step in each history. The step where a difference is detected is returned and recorded. The number of this step is passed to a collection of bug-recognition predicates, only one of which should succeed with a given pair of execution histories.

As indicated in section 6.3 the version of VIPER described here has been developed by concentrating on one of Brna et al.'s (1987) symptoms, that of a variable being instantiated to the wrong value. The 'bug trees' described in section 6.3 suggest that a specific group of bug recognisers can be assembled, based on the expectation that both goals will eventually succeed. Provision has also been made to incorporate bug recognisers based on the other symptoms defined by Brna et al. (1987): the unexpected failure to instantiate a variable, the unexpected instantiation of a variable, and the termination issues. This provision consists of a predicate which examines the results of the bugged and ideal execution to ensure that they are different, and to determine which symptom is being exhibited. According to the symptom exhibited, a collection of bug recognisers is chosen for the analysis of the execution histories.

At VIPER's present level of development, this symptom-recognising predicate always chooses the same set of bug recognisers, as the code being input ensures that the same symptom is always exhibited. These bug recognisers are based on the abstractions developed in section 6.4 which describe a given bug's effect on execution in terms of the success or failure of the resolutions being attempted at the point where the execution histories differ. Each bug recogniser is called in turn, and the one which succeeds is given as the output of the analysis. These bug recognisers are not intended to model human inferencing. They are intended to identify the correct answer given the conditions under which they are run. The recognisers have two kinds of subgoal: those which succeed to identify a given bug, and those which fail in order to prevent the recogniser succeeding erroneously. Recognisers are present for each of the legal possibilities identified in table 2 of section 6.4. This means that a single bug may require up to three different recognisers,

one for each of the execution patterns that the bug may manifest.

A range of tests are defined which may be used by more than one bug recogniser. These tests determine whether a given resolution is successful, or whether it exhibits one of a number of kinds of failure such as 'functor failure', where the functors do not unify or 'subgoal failure' where a subgoal does not succeed. These tests look for sequences of execution action symbols in the execution history facts which follow the first point where the two histories differ. Where necessary, the bug recognisers retrieve and compare parts of the two clauses which are being resolved with the goal at the point where the histories differ. This allows the recognisers to determine the exact difference between the ideal and bugged code. A range of predicates is defined to retrieve such features as functor, arity, or subgoals and subgoal order for a given clause.

7.3.3 Data available after execution history analysis.

After running the ideal and bugged code through the meta-interpreter and analysing the resulting execution histories, VIPER has the a range of information available to it.

The principal items of available information are:

- the input goal and databases;
- the result from the ideal code;
- the result from the bugged code;
- the ideal code execution history;
- the bugged code execution history;
- the execution step at which the two histories differ;
- the goal and clauses whose resolution is being attempted when the histories differ;
- the bug which causes the difference in execution.

The parts of VIPER which have yet to be described are the mechanisms which support the tutorial exploitation of this information.

7.4 The tutorial dialogues.

7.4.1 Introduction.

VIPER has been developed to a point where a number of quite distinct dialogues may be conducted between the system and the student. These dialogues are not intended, as they stand, to be highly adaptive to the user. Rather, their purpose is to demonstrate the various mechanisms that are available for use in tutoring, and which, it is claimed, could be used in a large number of ways to support adaptive tutoring. The point of VIPER is thus that it provides the *potential* for adaptive tutoring using multiple viewpoints on the domain, and that from this stage of development onwards, this potential can be realised using well-known methods.

This is consistent with the research direction described in chapters 2 and 3, which calls for an investigation of the issues involved in building a system which can tutor in terms of several pre-defined viewpoints. The major implementational effort has gone into producing mechanisms which can manipulate and explain viewpoints, rather than into mechanisms which can dynamically adapt the tutoring of them.

7.4.2 Using the procedural versions of the Prolog models.

The conclusions to chapter 6 outlined the pedagogical goal of VIPER. These state that the student should develop ability to describe Prolog execution in terms of the models defined in section 6.5, and that they should be able to use these models to localise the bugs in the simplified world of the bugged and ideal code. This implies that at least three kinds of dialogue are necessary. An obvious prerequisite of using the models to localise bugs is that a student can use them accurately to describe straightforward program execution. The first kind of dialogue thus simply asks the student to describe the execution of a particular query and database irrespective of whether it is bugged or not. When the models are being used

to localise the bugs, the first step must be to clearly identify the clause which contains the bug. The second dialogue is thus focused on this task. Having identified the correct clause, the third dialogue checks that the student has a correct understanding of the implications of the bug by asking them to describe its effect on the program's execution. At each stage, explanatory dialogues may also be necessary. The structures detailed in chapter 6 are used to implement the knowledge representations of a system which can support these dialogues.

Dialogue 1: describing execution.

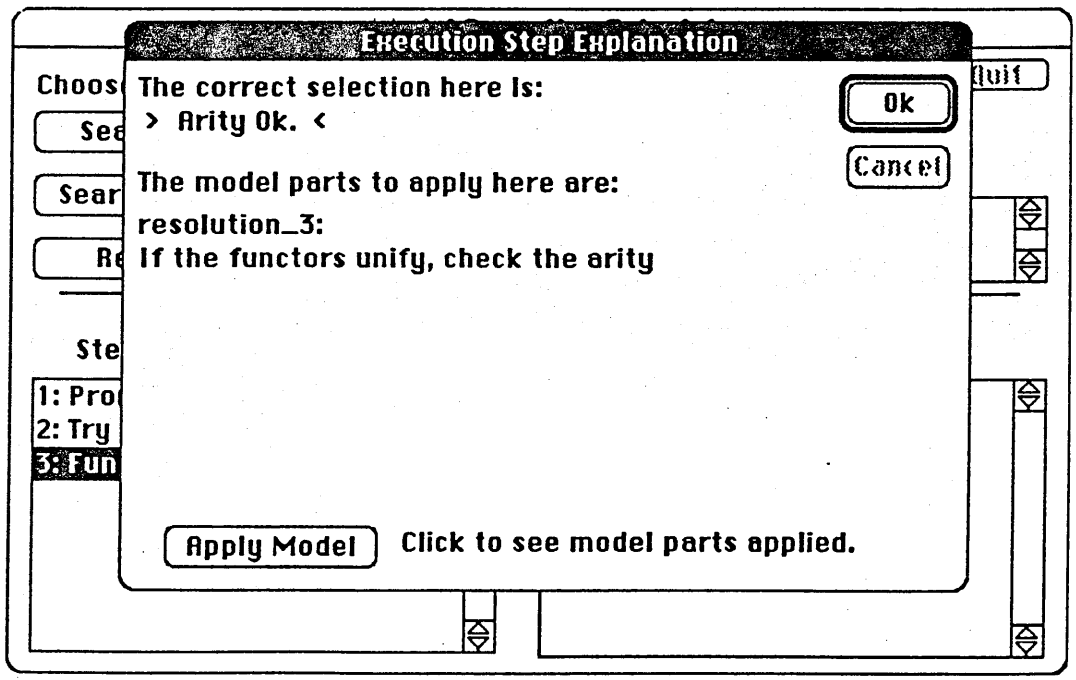
The first dialogue requires the use of the re-formulated model parts of section 6.5 to describe specific executions as a series of discrete steps. The model-parts are abbreviated so that the events covered by each one can be offered as a set of menu choices, each of which is associated with one of the action symbols stored in the execution history facts by the meta-interpreter. A student makes a succession of choices from the menus to describe a specific execution. The accuracy of these choices is checked by comparing the symbol associated with the menu choice with the symbol stored in a trace of the execution. The full range of menu choices and associated symbols is given in table 7, and an example of an on-screen menu is given in figure 15.

An explanation template is associated with each symbol. (These templates are related to the individual model-parts of section 6.5, and have no relation to the templates described in section 6.4 which describe the effects on execution of each bug). The template has slots for such values as clause and goal that can be filled by reference to the execution history facts and the initial database. VIPER can justify its assessment of a menu choice by stating which model part is relevant to the current execution step, and thus which menu option should have been selected. If further explanation is required, the slots in the relevant explanation template can be filled with data from the relevant execution history fact, and the resulting text presented to the student. VIPER can also demonstrate how an execution

should be described by using this explanation mechanism continuously. In this mode, VIPER does not explain a single step in reaction to input from the student, but fills and presents the explanation template relevant to each fact in succession. Figure 11 shows an example of an explanation presented during this first form of dialogue in relation to rule 'Resolution 3'. Figure 12 shows an example of the rule referred to in the explanation of figure 11 being applied to the execution step which was current at that point.

The three models of section 6.5 are interpreted as describing 19 possible situations or events, such as 'subgoal succeeds', or 'functors of goal and clause unify'. Menu options are available for each of these. One further option is provided for the statement that 'search is complete'. Two other symbols are present in the execution history facts which are not related to menu choices, but which are used by the bug recognisers described in section 7.3.2.

Figure 11. An example of the explanations used in Dialogue 1.

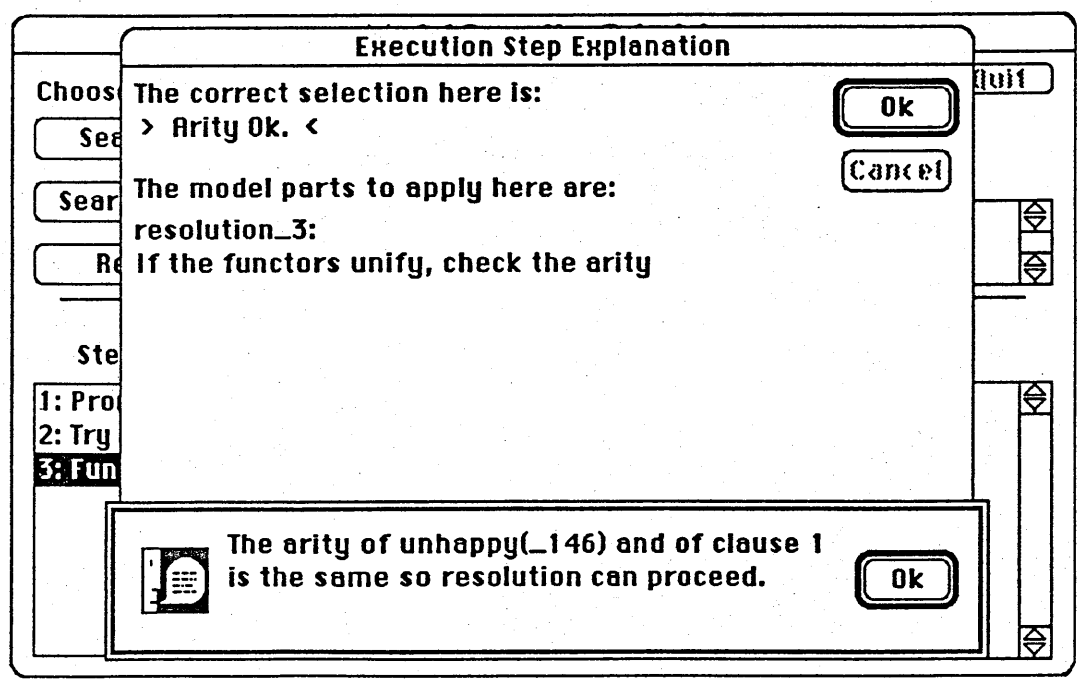


In summary, each rule/explanation-template combination is also associated with a specific symbol and a menu option. (See example in fig. 8). The symbols allow VIPER to check

students' input in relation to particular executions, and to compute suitable output for explanation.

This is exemplified in table 6. This describes the case where the functors of goal and clause unify successfully, through a combination of symbol, rule, explanation template. The correct menu choice in this case is "Functors Ok". This choice is also associated with the symbol '/'.

Figure 12. The explanation of figure 11 applied to the current execution.



The entry 'R1' for 'Rule' in table 6 refers to the procedural models given in section 6.5, where each part of each rule is given a label consisting of an abbreviation of the name of the model ('R' for Resolution, 'SST' for Search Strategy, 'SSP' for Search Space) followed by a number indicating the rule's position in that model. The italicised capitals in the explanation template are placeholders for values which may be instantiated from the relevant trace segment.

Table 6. The symbol, rule and explanation template combination for the execution event where the functors of a goal and clause unify.

Symbol.	'/'
Rule.	R1
Explanation	For the resolution of goal *GOAL* and clause *CLAUSE* the functors unify.

The pedagogical purpose of this exercise is that the students should learn which model to apply in order to describe the successive steps of an execution. Thus, before they can choose a menu option, they have to choose the model that is currently applicable. This is achieved by having buttons on the screen, one for each of 'Search Space', 'Search Strategy', and 'Resolution'. When a student clicks on a button, the relevant menu pops up. The selection made by the student is returned to a predicate which checks its accuracy and takes the appropriate action. An example of the screen for the execution description dialogue, showing the model selection buttons, is given in figure 14. An example of this screen showing one of the menus used to describe a step in the execution, is given in figure 15.

Since the student has to make explicit choices at two distinct levels, (the model and the model part), it is clear that this behaviour could be monitored and tutorial interventions made at either level. At present VIPER does not analyse the student's input in any complex way, but merely decides whether it is right or wrong at the level of the menu choice. If the answer is correct, it is entered into the evolving execution description in the dialogue window, and VIPER moves on the the next execution step. In the case of a wrong answer, the action taken depends on the setting of a somewhat basic student model. With the relevant setting 'on', VIPER responds to all wrong answers by informing the student that a mistake has been made, and by giving the correct answer. The student can click on a button to request further explanation. In this mode, wrong answers are not entered in the

execution description. This 'highly interventionist' mode of tutoring can be compared to the 'model tracing' used by Anderson and Reiser's (1985) LISP tutor, which does not allow the student to make a mistake. If the student model setting is turned 'off', then corrections are not given, and the menu selection is entered into the evolving execution description. The student model also keeps a numerical record for each model part of potentially correct, correct, and wrong answers. It is assumed that these selection records could be developed to support diagnostic analyses.

The full list of menu options and their associated symbols is given in table 7. (For each possible choice the VIPER also has available a rule and explanation-template combination).

The 'search complete' option of the Search Strategy model is not strictly a part of the model as given previously, but is provided so that a definite end point to the student's (or the system's) description may be stated.

The structures described in this section (Dialogue 1) can be summarised by saying that the procedural models of section 6.5 were abbreviated to give a series of menu options. Each of these is associated with one or more abstract symbols, an explanation template, and a rule. Figure 13 shows these structured relationships for the proceduralised Search Space model.

Table 7. The proceduralised models abbreviated to a set of menu choices, with their associated system symbols.

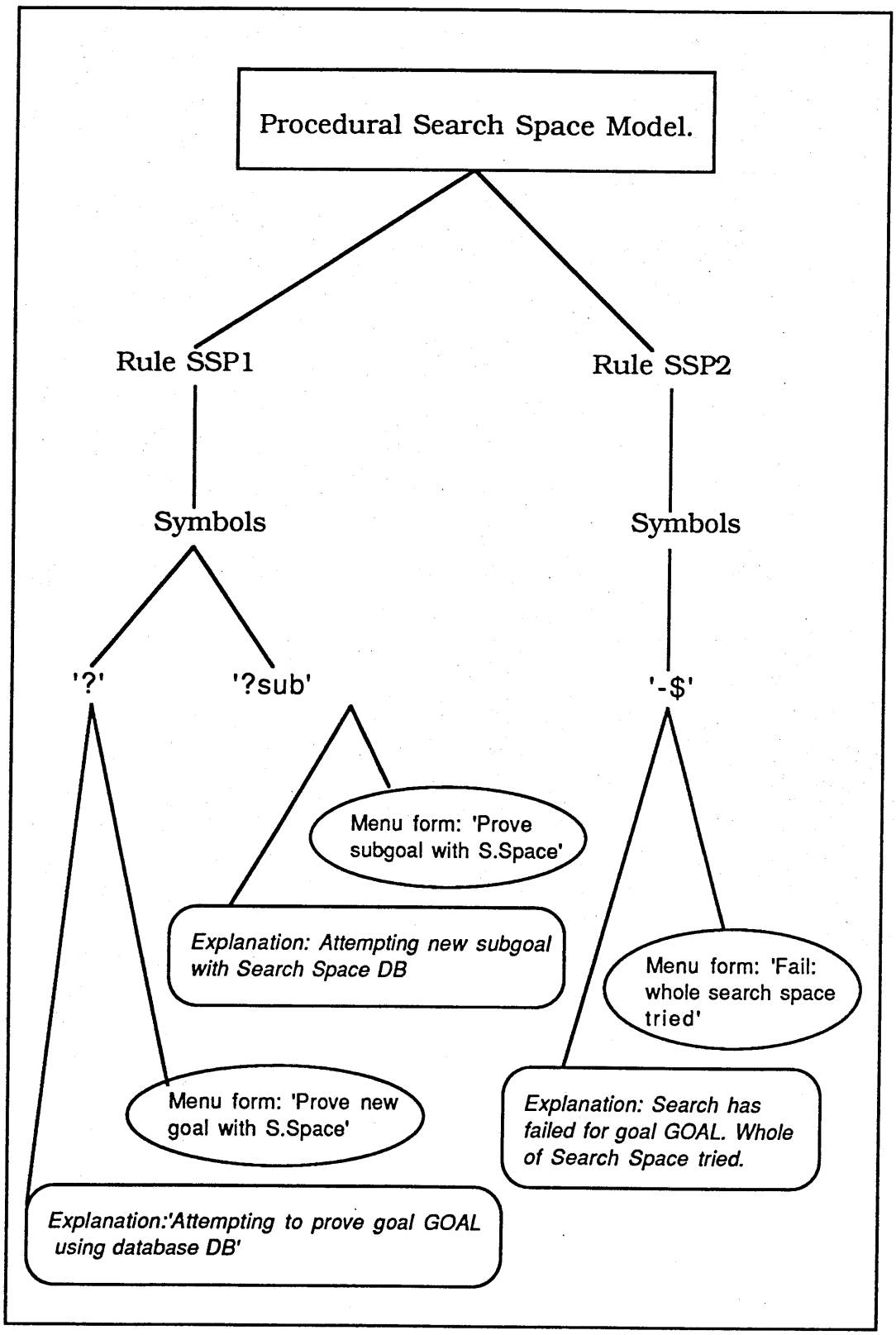
Menu Options and symbols for the 'procedural' Search Space Model.	
Menu Options.	System Symbol.
Prove new goal with search space.	'?'
Prove subgoal with search space.	'?sub'
Fail: whole search space tried.	'-$'

Table 7 Continued.

Menu Options and symbols for the 'procedural' Search Strategy Model.	
Menu Options.	System Symbol.
Try goal with next clause.	'>'
Quit this resolution.	'<'
Subgoal ok. Try next subgoal.	'+'
All subgoals ok, Parent ok.	'+***'
Subgoal fails: Parent fails.	'-n'
Proved on Fact: no subgoals.	'+*'
Resolves with head: try subgoals	'+**'
Search complete.	[]

Menu Options and symbols for the 'procedural' Resolution Model.	
Menu Options.	System Symbol.
Functors ok.	'/'
Functors fail.	'\'
Arity ok.	'≥'
Arity fails.	'≤'
Argument pair unify.	'u'
Argument pair fail.	'-u'
All arguments unified.	'v'
Goal with operator ok.	'Δ'
Goal with operator fails.	'•'

Figure 13. The procedural Search Space model with its associated symbols, explanation templates and menu versions of model parts.



As is clear with the rule 'SSP1' in figure 13, a single model part can have more that one event symbol associated with it. This is consistent with the content of the procedural model parts, which may cover more than a single possible event. (See statement of proceduralised models in section 6.5). In the case of SSP1, the possible events are firstly, the setting of the initial goal literal or query with the database, and secondly the setting of a generated subgoal to be proven with the same database. The implemented meta-interpreter and tutoring system recognise these as separate events which merit the use of separate menu choices and explanation templates to describe them. The other models also have multiple symbols associated with certain model parts.

The 'execution' dialogue described above is supported by a range of predicates. All the relevant model parts, menu choices and symbols are held in a system database, along with the names of predicates which instantiate and present the explanation templates when this is required. The dialogue window (see figure 14) contains the model selection buttons, a scrolling menu which displays the evolving execution description, and a display of the current goal in a text field. The window also contains two more scrolling menus. The first of these records any unifications which are made in the current resolution, and the second displays the full database for the current execution.

Each time a button is clicked, a predicate associated with the 'execution' dialogue is called, with the number of the button as an argument. The appropriate definition of the predicate generates a popup menu which returns the user's selection; (see figure 15). The same predicate which called the menu now calls another predicate to assess the selection which has been input. If it is correct, the various dialogue fields and menus are updated, and the next step is considered. If it is not correct, (and the relevant student model switch is 'on'), the correction and explanation mechanisms described above are brought into play. When the correction/explanation activities are completed, the call from the model button is made to fail, so that control returns to the original dialogue which cannot end until the goal associated with it succeeds.

Figure 14. The screen for describing code execution.

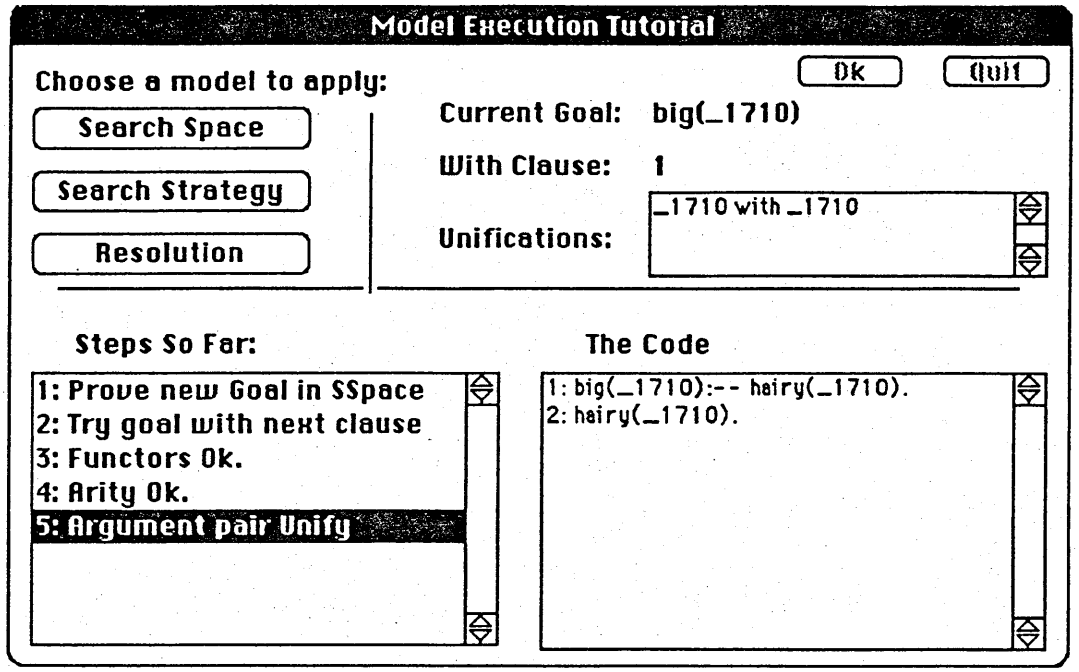
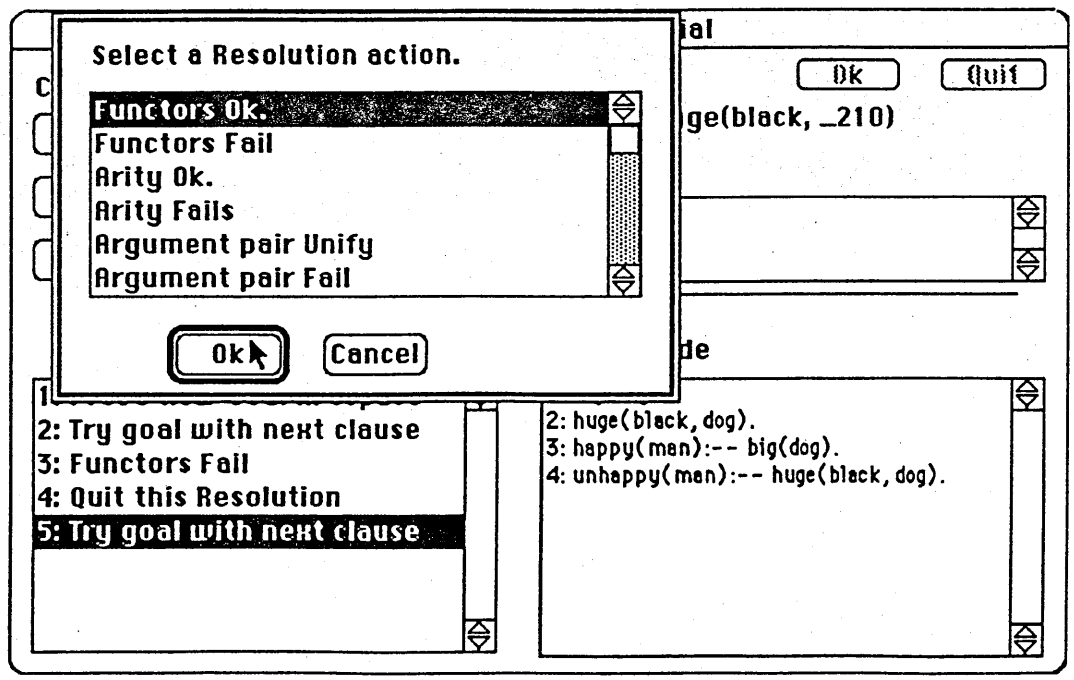


Figure 15. The screen for describing code execution showing the Resolution model menu 'popped-up' to take student input.



Dialogue 2: Identifying the bugged clause.

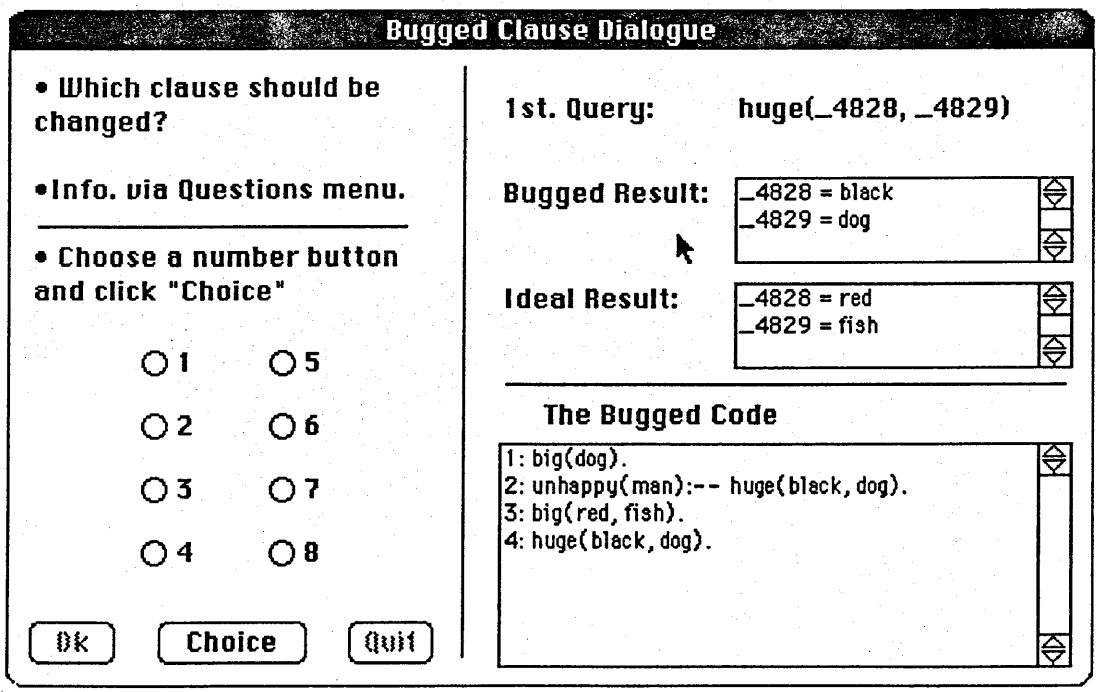
When it is clear that the user understands and can apply the models used in the execution dialogue, then a tutorial dealing with a specific bug may be started. (The decision about when to proceed is currently made by the student rather than by VIPER). To provide the information necessary for a tutorial, a set of query, ideal code, and bugged code is retrieved from the system database. At VIPER's current level of development, the student is asked to indicate what viewpoint they wish to work with, and a query-code set associated with that viewpoint is chosen at random. (The different ways of analysing execution are referred to as 'viewpoints' rather than 'models' in this second phase, as the models have to be used in conjunction with a set of inference procedures). The initial query is run with both the ideal and bugged code, and the resulting sets of execution history facts are analysed as described in section 7.3.2.

The second dialogue then begins. The screen for this dialogue (see figure 16) shows the initial query, the results obtained with the ideal and the bugged code, and the bugged database. The screen also shows a number of 'radio' buttons associated with the question "Which clause should be changed?". ('Radio' buttons indicate alternatives from which only one may be selected. When the user clicks on 'Ok', the number associated with the selected button is returned as an argument). These buttons represent the clauses of the bugged database, and each is associated with a number. (The database clauses are also numbered).

In VIPER's current state, this dialogue will not terminate until the number representing the bugged clause is selected. (Under the conditions laid down for this system, VIPER identifies the bugged clause as the first one which fails to match its corresponding ideal database clause). The student is not left to blindly guess the result. At the start of this dialogue a pull-down menu is installed which enables the student to ask questions about the ideal database; (see table 8). This is intended to allow the student to test hypotheses which

they may have formed concerning the nature of the bug, and the identity of the bugged clause. This menu also allows them to propose candidate bugs as being responsible for the results shown, and to ask for a complete explanation of the bug should this be desired.

Figure 16. The screen for identifying the bugged clause.



This 'Questions' menu is intended to facilitate and encourage the use of the kinds of inference operators described in chapter 5. The explanations that are provided through the menu are structured around versions of the "inference" operator. The options available on the menu enable the correct line of inferencing to identify whichever of the bugs is present from the list given in chapter 6. Where necessary, hierarchical menus allow the user to make a general question specific. (The list of menu options is given in table 8, along with an indication of the contents of the hierarchical popup menus). Thus the question 'what is the functor in clause ...' is supported by a popup hierarchical menu which lists numbers from 1 to 10. By selecting a number, the user asks to be told the functor of a specific ideal database clause. As this operation simply retrieves information which is already explicitly

stated, it is seen as equivalent to the use of the "type 1" of "access" operator described in chapter 5.

This data could be used, for example, to determine whether the hypothesis that the bug is a 'wrong functor' in clause *N* is correct. If the hypothesis was correct, the explanation offered by the system would be structured around the 'type two' operators discussed in chapter 5. These were described as combining two or more parts of a model via an inference procedure to produce a previously unstated conclusion, without changing the model, (eg. A -> B, B-> C, therefore A -> C). The notion of 'model' as being the thing which is interrogated by operators must here be broadened to include both the execution history which results from the description of a specific execution in terms of the models, and the concept of a 'bug'. (This reflects the actual situation in the study described in chapter 4, where the subjects were able to observe an on-screen simulation of a system as well as to refer to the models as they were stated on paper). The viewpoints cannot be used by the system to localise bugs in actual executions unless this is done, as the models of execution only describe possible executions in abstract terms. The execution histories are structured in terms of the models of execution, and thus constitute an application of those models to describe actual executions. Neither VIPER nor the students using the system could use the viewpoints to localise bugs unless an inference procedure was available which linked the models to the execution histories.

In the broader context the operator is applied to produce the following kinds of inference:

step 1: Resolution rule 1, and functor X in bugged database clause N implies that

step 2: a goal resolves with clause N and the wrong variable binding is made, implying that

step 3: clause N in the ideal database has a different functor.

The relevance of this to the question menu can be seen when the conclusion of the operator is stated, ie. that step 1 implies step 3.

In the terms stated above:

step 1: Resolution rule 1, and functor X in bugged database clause N

implies that

step 3: clause N in the ideal database has a different functor.

This conclusion can be easily checked using the 'Questions' menu.

Another version of the operator is defined for the case where the goal and clause *failed* to resolve, so that the desired variable binding was not made. Again, the conclusion is that the ideal code clause N has a different functor, and this can again be checked using the 'Questions' menu. The notion of an operator is thus used to encode the kinds of inference that can be usefully made in terms of the three models when they are applied to a specific execution. Explanations in these terms are provided for all of the allowed bugs, and all of the conclusions can be tested through the 'Questions' menu.

As yet VIPER makes no attempt to tutor these operators directly, other than providing explanations structured around them. It is, however, interesting to speculate on what could be achieved by monitoring the student's use of the 'Questions' menu to determine the extent to which the questions being asked were relevant to the bug at hand, and to determine the accuracy with which they were applying the kind of operators discussed above. What is proposed here is that these operators are a useful way of capturing the distinction between simply applying the models to describe an execution, and applying them to localise a bug.

These menu options are supported by predicates which retrieve the ideal code from the system database, compute the requested information, and display it to the student. The accuracy of the code error suggested by the student is checked by consulting a database of relevant information which is stored after the two sets of execution facts have been analysed.

Table 8. Options available on the 'Questions' menu.

Menu Option	Contents of Hierarchical Menu
'What functor in clause...'	Numbers 1 - 9.
'What arity in clause...'	Numbers 1 - 9.
'What arguments in clause...'	Numbers 1 - 9.
'What subgoals in clause...'	Numbers 1 - 9.
'Show clause...'	Numbers 1 - 9.
'Is the bug...'	List of legal Bugs.
'How many clauses in ideal'	None
'Give Explanation'	None

The explanations which are requested from the 'Questions' menu (and which are structured around the 'inference' operators described above), are assembled by predicates which carry out a different analysis of the execution history facts. These predicates are in this sense 'free-standing' in that their selection of material to present to the student does not depend on the original analysis carried out by the 'bug recognisers' described in section 7.3.2. This involves some redundancy in the system, as it would have been a simple matter to define a range of explanation templates, one of which could have been selected in accordance with the bug detected by the bug-recognisers. The justification for the redundancy is that the predicates defined to provide the explanation have different purpose and operate in a different manner. These predicates will be referred to as the 'bug-explainers'.

The bug-*recognisers* were implemented to carry out their function in the most computationally efficient way that could be arranged in the given system, even though this

may have little relation to human inferencing in a similar situation. A recogniser is built for each of the combinations of success and failure in the ideal and bugged databases which are described in section 6.4.2. These recognisers simply identify specific sequences of symbols at the same points in the execution histories of ideal and bugged code, and then test for specific differences between the ideal and the bugged code. Although this is a 'black box' solution, we would wish to claim that it can still support useful tutorial activities. (This issue was discussed in relation to Burton and Brown's [1979] WEST system in chapter 2).

The bug-*explainers* are intended to simulate (as far as possible) the reasoning of human investigators in the manner described for the "inference operators" of chapter 5. As in the example of a wrong functor given several paragraphs above, the bug-explainers assemble the information about the bugged code and the bugged execution that might be used by a student, and test the conclusion that is implied by this information; (ie. that there is a specific difference between the ideal and bugged code). If the conclusion is found to be true, the bug-explainer succeeds and its information is placed in the slots of a text template for presentation to the student as an explanation of the bug.

When an explanation is requested, all available bug-explainers are called in turn. This implies that only the correct one should be allowed to succeed if the explanation is allowed to be relevant. In order to prevent irrelevant successes, some of the bug-explaining predicates have to be given extra subgoals which are not strictly related to the inferencing required by the operator.

The bug-explainers differ from the bug-recognisers in another respect. A bug-recogniser is defined for each of the combinations of success and failure in ideal and bugged database described in section 6.4.2. (Generally, at a given point in the execution, a given goal may succeed in the bugged database where it fails in the ideal, may fail in the bugged where it succeeds in the ideal, or may succeed at the same point in both). The bug-explainers do not

discriminate so finely. At a given point in the execution, they are concerned only with the success or failure of the goal in the bugged clause. This is related to the applicable rule or modelpart, and the conclusion or 'hypothesis' concerning the ideal code is tested. Thus in general terms, a given bug will have two bug-explainers related to it. One deals with an unwanted variable binding caused by success, the other with an unwanted variable binding caused by the failure of a previous resolution. Each bug-explainer calls its own text explanation template. This is the third form of text template used by the system. It is not related to the templates discussed in section 6.4 which describe the effects on execution of each bug, nor to the templates described in section 7.4.2 (Dialogue 1) which explain the application of a specific model part at a given point in an execution.

The arguments for 'glass box' systems presented in chapter 2 may lead us to prefer the use of the bug-explainers to that of the bug-recognisers. However, the fact that the bug-explainers operate independently does not necessarily indicate that the bug-recognisers are wholly redundant. It is not clear at this stage that the bug-explainers are powerful enough to carry out an accurate analysis of the execution histories. Also, since the bug-recognisers encode all the legal combinations of success and failure that may occur under the conditions described in chapter 6, they constitute a very useful test of any input bugged and ideal code sets, since the execution histories produced by the code sets should cause one of the bug-recognisers to succeed. Such a test would be especially useful in an authoring interface, where code sets may be input by someone other that the system designer.

Dialogue 3: Describing the bug's effect on execution.

As indicated earlier, it is desirable that the student should do more than simply identify the bugged clause. In order to ensure that their understanding of the bug's effect is correct, they are asked to describe the execution of the relevant goal with the bugged clause in terms of the three models given in section 6.5. A new dialogue is initiated to carry out this activity. This dialogue is structured around the mapping from models to bugs that is

described in section 6.4.

The dialogue presents a screen that shows the relevant goal and the bugged code (see figure 17). Also on the screen are a sequence of clickable buttons labeled Functor, Arity, Arguments, Search Strategy, Search Space and Code Error. The student gives VIPER input relating to each of these aspects of the resolution under consideration by clicking on a button and making a selection from the menu that pops up as a result. These inputs need to be given in the correct sequence, so usually all but the currently relevant button are disabled. This ordering is imposed on the inputs in order to avoid incoherent combinations, such as the statement that the functor unification fails, and that the arity test succeeds. (According to the Resolution model, if the functor unification fails, the arity test is not made). If the student gives an input which is logically incompatible with the previous inputs, then an explanatory message is presented stating why that input is not suitable, and the dialogue returned to its pre-input state. If VIPER accepts the input, then it is entered into the dialogue screen next to the relevant button, and the screen updated ready for the next input selection. The possible input choices for each button are given in table 9.

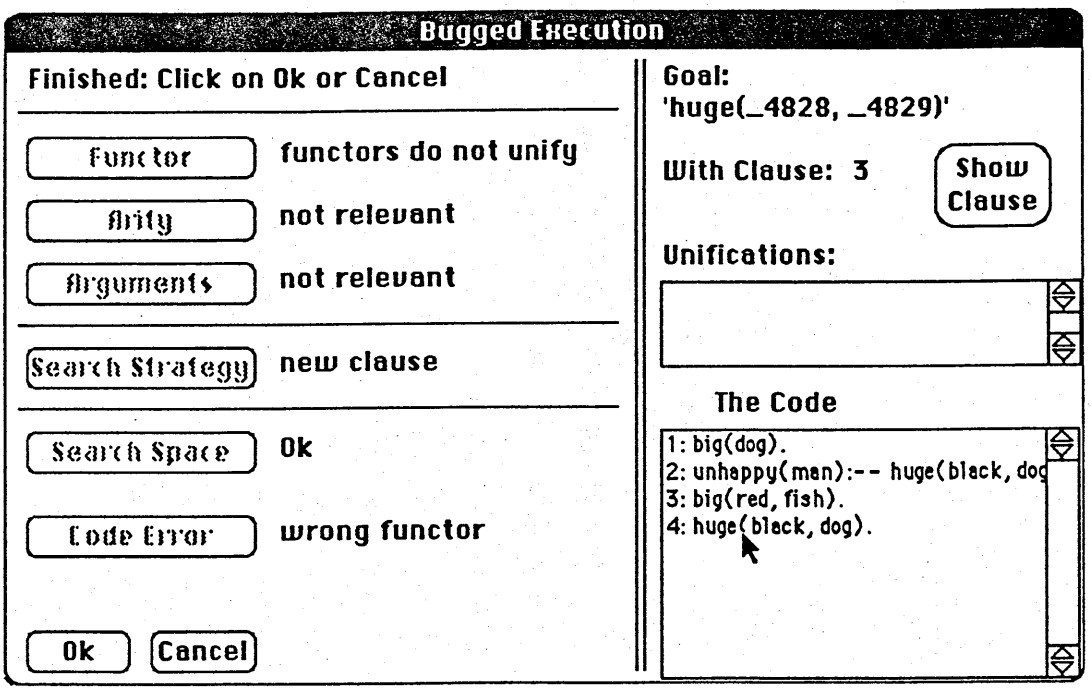
The possible entries for Functor, Arity, and Arguments summarise the result of applying the Resolution model of section 6.5 to the resolution which is being examined. The entry for Search Strategy states whether the search continues elsewhere, pursues subgoals successfully or unsuccessfully, or halts at the point being examined.

The Search Space entry requires a statement that the Search Space is either 'Ok', or else embodies one of the Search Space bugs. This accords with the conventions stated in section 6.4.3, which require that only bugs to do with the presence or absence of clauses or subgoals can be Search Space bugs. If the Search Space does not contain a Search Space bug, it must be 'Ok'.

The final choice to be made is that of the Code Error. If the Search Space contains a bug,

then the description of this will be identical with the description of the Code Error. Otherwise, the name of one of the other legal bugs should be entered.

Figure 17. The screen for describing the bugged execution.



At this point the student will have described the effects of the bug in terms of the three procedural models of section 6.5, and stated their perception of what the current bug is. They will not have made any explicit statement concerning the binding of variables to give the output symptom that the legal bug library is related to. Such statements could be added as another feature of the dialogue, and would relate to the issue, (explored in chapter 6), of which question about the bug is being asked; (ie. does the input given in this dialogue describe the case where a success causes the wrong variable binding to be made, or the case where an unwanted failure causes a binding to be made elsewhere).

Table 9. The input options available for each button of the 'Bugged Execution' dialogue.

Button.	Options Available.
Functor.	functors unify functors do not unify
Arity	arity same fails not relevant
Arguments	succeeds fails not relevant
Search Strategy	new clause. no subgoals: stops. subgoals: succeed. subgoals: fail.
Search Space	'Ok' + list of Search Space bugs.
Code Error	List of legal bugs.

The student's input could be analysed in terms of its internal consistency, and its relationship to the actual execution histories. Such an analysis could go well beyond the simple correctness or otherwise of the answers, to assess the students understanding of the models and their mapping onto the bugs and bugged execution. At its current level of development VIPER makes no attempt to analyse the students' inputs in this way, and does

not allow students to revise their choices once they have been entered into the dialogue screen. When a student has made choices in relation to all of the buttons, a click on an 'Ok' button ends the dialogue and gives VIPER all of the input choices. VIPER then simply states what the correct choices were. The 'Questions' menu, with its 'Get Explanation' option, remains available throughout this dialogue.

The correct choices to describe the execution of each bug are held in memory as a 'working frame' the details of which are assembled by the predicate which initiates the bugged execution dialogues. Because each of the bug-recognisers identifies a unique success-failure combination, it is possible to store an answer template related to each one, and to use this to provide the correct answers for describing the execution of each bug. If necessary, two templates are stored for a bug-recogniser. The first template describes the first goal/clause resolution in the bugged execution history which does not match the corresponding one in the ideal execution history. If the bugged resolution fails no variable binding can be made, and the second template will be required to describe the subsequent resolution where a variable *is* bound. This issue is described in more detail in section 6.4.3.

The templates referred to in the previous paragraph are stored as a sequence of codes. When the instantiated 'working frame' is being assembled, the template with its list of coded entries is retrieved. The definition of each coded entry is then retrieved and entered into the appropriate 'working frame' slot. In the case of the Search Strategy slots, some further computation may be needed, to determine whether there are subgoals to be proved or not, and if any are present, whether or not they succeed.

7.5 Conclusions to Chapter 7.

This chapter describes the implementation of a system designed to tutor Prolog novices using the abstractions described in chapter 6. This system, VIPER, is based on a meta-interpreter which, when run with a query and suitable code, produces a detailed execution history in terms of the three procedural models of section 6.5. This history is used to facilitate three forms of dialogue between system and student. The dialogues are concerned with describing execution, identifying the bugged clause, and describing the effect of the bug on execution.

Section 7.2 describes in outline the meta-interpreter which reproduces a subset of Prolog behaviour. This meta-interpreter takes a query and database as input, and produces a detailed history of their execution in terms of the three procedural models of section 6.5. The meta-interpreter is structured around the Resolution, Search Strategy and Search Space models of chapter 6 and records an execution history in terms of these models.

Section 7.3 describes the mechanisms by which the execution history is recorded and analysed. The execution histories of the ideal and bugged databases are compared to determine the point at which they differ. A set of bug recognisers are then run in sequence to determine the exact nature of the bug. A summary is given of the information available to VIPER when this analysis is complete.

Section 7.4 describes the three kinds of dialogue that the system can presently support. These dialogues are not intended to stand as examples of adaptivity to the user. Rather, their purpose is to demonstrate the various mechanisms which are available in VIPER, and which could be used adaptively. WE thus claim that VIPER provides the *potential* for adaptive tutoring using multiple viewpoints on the domain, and that this potential can be realised using established methods. The major implementational effort has thus gone into producing mechanisms which can manipulate and explain viewpoints, rather than into

mechanisms which can dynamically adapt the tutoring of them. We indicate where adaptive extensions to the system would be possible.

The first form of dialogue describes the execution of a query and database. This description may be made by the student or by VIPER. The facts of the execution history are related to menu choices made by the student. The options on the menus summarise the events covered by the rules of section 6.5. Inaccurate choices are corrected, and, if requested, a demonstration is given of the relevant rule being applied. VIPER can describe the execution by giving a series of these demonstrations consecutively. The purpose of this dialogue is to ensure that a student understands and can apply the three models accurately to describe execution.

The second dialogue asks the student to identify the bugged clause. Information about the ideal code can be gained by using a 'Questions' menu which can provide all the information that is required to differentiate the various bugs that may be present in the bugged code. The intention is that students should use this facility to test their hypotheses concerning the bug. If requested, explanations of the bug will be provided by mechanisms based on the 'type two' or 'inference' operators described in chapter 5. These demonstrate how the application of the procedural models of section 6.5 to the execution history can be combined with inference procedures to produce testable conclusions about the ideal code.

The third dialogue checks the student's understanding of the bug's effect. They are asked to describe the execution of the relevant goal with the bugged clause in terms of the three models given in section 6.5. This dialogue is structured around the mapping from models to bugs that is described in section 6.4. Students make a sequence of menu selections which summarise the resolution of the relevant goal with the head of the bugged clause, and the subsequent state of the Search Strategy. The students are then asked to state what defects, if any, are present in the Search Space, and to state the nature of the Code Error. VIPER's own analysis provides the correct answers.

Chapter 8. Evaluating VIPER.

8.1 Introduction.

The implementation described in chapter 7 supports three kinds of dialogue with the student. These involve describing the execution of a given query and code, (Dialogue 1), identifying the bugged clause, (Dialogue 2), and describing the effect of the bug on program execution, (Dialogue 3). As previously indicated, these dialogues are not intended to stand as examples of adaptivity to the user. Rather, their purpose is to demonstrate the various mechanisms which are available in VIPER, and which, with some augmentation, could support adaptive tutoring. The system thus claims to provide the *potential* for adaptive tutoring using multiple viewpoints on the domain. It is also claimed that this potential can be realised using well-researched techniques.

Before making any claims about the *adaptive* potential of the mechanisms, it is necessary to show that these mechanisms themselves are effective as they stand. "Effective" is here assumed to mean that the mechanisms can support a tutoring interaction that is deemed to be useful or effective by the human users who engage in it. The central assumption is that different viewpoints will help to localise different bugs in the code. The purpose of the system is to tutor the skill of applying the different viewpoints in a simplified Prolog environment, so as to localise different bugs.

The purposes of the evaluation were thus:

- To check that the components of the system functioned properly together; (ie. does VIPER run?).
- To test the usefulness of the way in which the different viewpoints have been encoded in the system; (ie. can VIPER exploit them usefully?).
- To test the overall design of the system; (ie. can VIPER carry out any useful tutoring, and what are its limitations?).

- To assess the usefulness of the viewpoints to Prolog beginners; (ie. are these useful irrespective of the system's effectiveness?)

This chapter reports the evaluation of VIPER by seven students of Prolog. Some of these were Prolog novices, and some were more experienced. They were given preparatory materials to read, introduced to the system interface, and each offered the same basic tutorial. The length and content of this tutorial varied according to the amount of practice they requested, and which model of simplified Prolog execution they chose to concentrate on at different decision points. On completion of the tutorial the students were asked to complete a questionnaire. The results of the evaluation are based on the responses to the questionnaires.

The evaluation thus had four parts:

- Three different viewpoints on Prolog execution were presented on paper so that those taking part could become familiar with them.
- Using the system, the participants applied the viewpoints to describe the execution of a specific 'Prolog' query and database. If they were required, the system provided demonstration, explanation, and corrections during this exercise.
- When the participants could confidently apply the viewpoints to straightforward Prolog execution, they were asked to apply the models to solve some simple debugging problems.
- When the debugging exercises were complete, the participants were asked to complete a short questionnaire about the system.

8.2 Method and Materials.

8.2.1 A tutorial with VIPER.

VIPER was run under LPA Prolog on a Macintosh SE/30. The users who took part were a combination of visitors, research students and research staff in a university department.

VIPER was configured so that each user would be presented with the same introduction and the same subsequent choices. The purpose of the introduction was to ensure that the users understood the interface for Dialogue 1, the execution description dialogue, and could confidently apply the models of section 6.5 to describe the simplified Prolog execution. Users were instructed to start the system when they were familiar with the simplified Prolog models. These were the models of section 6.5, and were printed on paper.

The introduction consisted of the following phases:

- Introduction to the task of Dialogue 1;
- Familiarisation with the interface;
- Demonstration of execution description by VIPER;
- Description of a very simple execution by the user:
- Description of a more complex execution by the user:
- Choice by the user of either:
    a) more execution description practice, or
    b) progression to part 2 of the tutorial.

If a user opted for more practice at describing code execution, the system presented them with a new query/code combination and asked them to begin a new execution description. When this description was complete, they were again presented with the choice of further practice or progression to part 2. This cycle continued until the user chose to go on to part 2.

While the user was becoming familiar with the interface for Dialogue 1, VIPER did not correct the menu choices they made to describe a simple code execution. After the system had demonstrated how execution was to be described, user input was always assessed and, if necessary, corrected.

When the user began part 2, the new task and the new interface were introduced. This task required them to identify the bugged clause in a database. The users were told that the bug constituted the single allowable difference between an ideal version of the code which was not visible, and the bugged version which they could see. They were also told that all the tasks in the second part of the tutorial related to the symptom 'Variable instantiated to an unexpected value'. They were informed of the presence of the 'Questions' menu, and told that it could provide information about the ideal code, information about the bug, and a full explanation of the bug and its effect. Users were then shown a list of bugs and told that each bug related to a specific viewpoint.

Dialogue 2, on its completion, leads straight into Dialogue 3. Users were thus informed about the structure of Dialogue 3 before beginning the tutorial. They were told that they would be shown the wrong or 'bugged' result, the correct or 'ideal' result, and, after they had identified it, the bugged clause. They were told that they would then be asked to describe the execution of a specific goal with the bugged clause at the point where it first differed from that of the same goal with the 'ideal' code.

It was stated that this involved:

1) Describing the resolution of a specific goal with a specific clausehead;

2) Making statements in relation to the Search Strategy, the Search Space, and the Code Error.

Users were then asked to select which model, (Resolution, Search Strategy, Search Space), they wished to concentrate on. VIPER then retrieved a set of bugged and ideal

code and appropriate query, the bug being related to the chosen model. (See section 6.4.3.2). The code was run through the meta-interpreter so that the execution histories could be recorded and Dialogue 2 was begun.

Dialogue 2 concluded with the correct identification of the bugged clause. At this point, they were reminded that the next task involved describing the bug's effect on the execution, and Dialogue 3 began. When the student had provided input to all the slots of Dialogue 3, no comment or assessment was made, but the correct answers were displayed.

Users were then asked if they wished to work with another bug. If the answer was positive, they were again asked to choose one of the models as the focus of the work, a suitable new code set was retrieved and run, and a new iteration of Dialogue 2 was begun. A negative answer ended the tutorial.

Users could keep selecting models and working on the problems that VIPER retrieved for them until they decided to stop.

8.2.2 Printed Materials.

Users always had access to the following printed materials:

- A briefing document describing the purpose and structure of the evaluation.
- A printed version of the three models detailed in section 6.5.
- A printed version of the menu choices related to each model which were used to describe execution.
- A printed version of the queries and databases used in Dialogue 1.
- A listing of the possible bugs related to each viewpoint.

These materials are given in appendix 2.

The queries and code used in Dialogues 2 and 3 are given in appendix 3.

8.2.3 The Questionnaire.

The questionnaire contained 17 questions organised into 5 sections, and is given in full in appendix 4.

The sections dealt with the following topics:

1. Experience of Prolog.
2. The Interface.
3. The Viewpoint Representations
4. The system.
5. The Viewpoints.

These sections were intended to reflect the main preoccupations underlying VIPER's design and implementation. Section 1 was intended to determine the level of the user's experience of Prolog. Section 2 enquired about the user's experience of VIPER's interface. The purpose of this was to check that the Macintosh interface and small monitor had not seriously inhibited their use of the system.

Section 3 focused on the system's representations of viewpoints and the use made of those representations. The questions in this section were intended to determine the usefulness both of the structures developed in chapter 6, and their implementation in VIPER as described in chapter 7. Questionnaire section 4 was concerned with more strategic questions such as the usefulness of adding diagnostic capabilities to the system, the usefulness of the system's ability to focus on a specific viewpoint, and the degree of learning that had taken place during the tutorial. Section 5 related to even more general issues. The first question asked whether the specific viewpoints that had been used were useful (without reference to VIPER). This was intended to check that the viewpoints formulated in chapter 6.5 were indeed useful. The second question asked the user to

suggest any other viewpoints that might also relevant with a view to possibly incorporating them in future developments of the system.

The report of the results from the questionnaires is structured in terms of these five sections.

8.3 Results.

The results reported here are a summation of the responses of 7 users to the questionnaire described in section 8.2.3. These results are structured in terms of the five sections of the questionnaire.

8.3.1 Responses to Questionnaire section 1, experience of Prolog.

Questions 1.1 and 1.2.

- How long have you been learning Prolog?
- How would you rate your ability in Prolog? (poor, fair, middling, good, very good).

The majority of users assessed their Prolog ability as being in the middle range, with two entries each for "fair", "middling", and "good". There was one entry of "poor". The length of time that they had been learning Prolog did not vary consistently with this assessment. The "poor" user had been learning for three months. One "fair" user had been studying Prolog for a few weeks, the other for a few months. One "middling" user had a year's experience, while the other had ten weeks. Both of the "good" users had studied a ten week course.

Question 1.3.

- Did you have a clear model of Prolog execution before this session?

Five users felt that they had had a clear model of Prolog execution before starting the tutorial. The user with "poor" ability stated that he did not have a good model at this time, while one "good" user stated immediately that his understanding of Prolog was clearer after using VIPER.

8.3.2 Responses to Questionnaire section 2, the Interface.

Question 2.1.

- Did any parts of the interface not function correctly?

Four users reported that they found no problems with the functioning of the interface, although five users criticised the behaviour of the menu which displayed the choices made in Dialogue 1, (describing the steps of a specific execution). This was a scrolling menu, which meant that when more than eight choices had been made, they could not all be viewed at the same time. Each time a new execution step was added, the menu was re-displayed with the first rather than the latest steps visible. The user thus had to scroll to the bottom of the menu to see the latest sequence of steps each time a step choice was made. While this was irritating, the LPA Prolog environment provided no obvious means of correcting the behaviour. The menus this environment provides are generally designed to take input rather than to display output.

A more serious bug caused the tutorial to terminate prematurely if certain sequences of viewpoints were chosen as the focus for work in Dialogues 2 and 3. This was due to characteristics of the LPA environment which affected the variable identifiers used in the meta-interpreter and caused the executions of this interpreter to fail when they should have succeeded. When different sequences of viewpoints were chosen, or when the LPA system tracer was switched on, the behaviour did not occur. Two users reported this bug. When it occurred, part 2 of their tutorial was restarted.

One user was confused by the similarity of two problems set to him after choosing the 'Resolution' option twice running at the start of Dialogue 2. He believed that the problems were the same, although this was not in fact the case.

Question 2.2.

- Did you find any parts of the interface difficult to use? (buttons, menus, etc.).

The majority of users reported that the interface was easy to use, although a number of criticisms were made. Two users commented that the buttons were very easy to use.

Two users were not familiar with the Macintosh interface, and were initially slowed down by this. They quickly became familiar with it. Two users reported some confusion over the fact that a scrolling menu for input could contain options that were not always visible. One of these was unfamiliar with the Macintosh, while the other suggested that all options on a menu should be visible at the same time. This suggestion was tempered by a recognition of the need to compromise on the use of screen space.

A number of other comments were related to the lack of screen space. One user stated that wider menus would have been preferable, while another pointed out that when menus were used to display the database as an aide in Dialogue 3, the ends of longer clauses were sometimes invisible as they ran off the side of the screen.

The screen size also meant that explanation and correction appeared in a window which overlaid parts of the main display of the current Dialogue. Some of the detail that the explanation or correction referred to was thus obscured as the explanation was delivered. This was criticised by one user. The same user also found a small problem with the hierarchical menus. When the top-level (pull-down) menu item was selected, a sub-menu appeared for as long as the mouse button was depressed. The final selection was made by moving the cursor onto the sub-menu and releasing the mouse button when the desired item

was highlighted. The user found that in moving onto the sub-menu, he sometimes unintentionally moved down one option on the main menu.

Another user complained about the number of mouseclicks required in Dialogue 3. He suggested that for each stage, the relevant menus should appear automatically, rather than the user having to click on the relevant button to call up the menu.

Question 2.3.

- Did you find that any part of the interface was particularly useful?

Responses to this question were generally quite complementary, with users indicating that all of the interface was relevant and useful, especially the frequent use of menus. Two aspects of the interface were given a special mention. One user stated that "...the 'Questions' menu was well-designed and encouraged browsing", while another reported that the display of the unifications made for each clause and goal in Dialogue 1 was particularly useful.

Other remarks were ambiguous as to whether they related to the interface or the underlying system architecture. One of these praised the part of the system relating to bug-finding, which is taken to mean Dialogue 2, while another stated that the "help" or "error-explanation" facility of Dialogues 2 and 3 was particularly useful.

Question 2.4

- Is there anything that you think should be added to the interface?

A number of possible additions were suggested here. One user was content with the interface as it stood, while another simply requested a bigger screen so that his interaction with it would involve fewer button clicks. The issue of screen size was also relevant to a point made about overlapping windows in responses to question 2.2. The same user who objected to their work being overlaid by an explanation also suggested a possible remedy.

This involved a facility for switching the explanation window into the background, so that a user could look at the tutorial screen with the explanation in mind, and then return to the explanation if desired.

Other suggestions related to specific Dialogues, (1-3). Two useful points were made about the interface for Dialogue 1. One user wanted some form of trace of the execution state, independent of that being developed by the user. The purpose of this was to help them to remember the overall state of the execution, this information being easily forgotten as they concentrated on the detailed analysis required by Dialogue 1. As this user was aware of the constraints of screen space, they suggested a remedy which would cost little. This involved underlining or highlighting the database clause that was currently being resolved with a goal. A different user suggested that when VIPER corrected an execution step, it should also enter the correct answer into the display itself, rather than waiting for the user to actively select this option.

Another suggestion was made for the Dialogue 1 interface which is more directly relevant to the pedagogical goals of VIPER. This suggestion stated that in the description of the execution, the model relating to each execution step should be clearly identified, and that the changes from one viewpoint to another should be *very* clearly indicated by highlighting or by sound. It was specifically stated that when the Resolution model was being applied, the choice of model-part "all arguments unify" should also clearly indicate that the Search Strategy model became immediately applicable. These suggestions were intended to give more structure to the list of execution steps displayed on the menu.

This same user made some equally relevant suggestions for Dialogues 2 and 3. The suggestions for Dialogue 2 were that the window title should state what viewpoint, and thus what kind of bugs, were the focus of the current work, and that a 'list of possible bugs' should be available for that viewpoint. These should prevent the user forgetting what viewpoint they were concentrating on, and thus straying to consider unrelated bugs.

The suggestion for Dialogue 3 was that when VIPER displayed the correct answers, it should give a 'right/wrong' appraisal of the user's input. A different user suggested that, having given some answers to the slots of Dialogue 3, they should be able to go back and change them if those answers were later considered to be mistaken.

One user who became particularly enthusiastic about the system turned out to be an expert in the field of natural language generation. This user provided some lengthy and detailed suggestions about how the techniques of this field could be applied to enhance VIPER's output. As these mainly relate to mechanisms for diagnosis and explanation, they will be dealt with as responses to question 4.1.

8.3.3 Responses to Questionnaire section 3, the Viewpoint Representations.

Question 3.1.

- Please comment on the way in which the system described execution, (ie. in terms of three viewpoints, each composed of a set of rules).

The comments on the way that the system described execution were generally very positive, with some minor caveats. One user commented that the description was very "rigorous and clear", while another stated that it should help beginners to understand Prolog execution clearly. A different user commented that the description, while generally good, did not clearly present the recursive nature of Prolog. This user reported an occasional confusion about which model should be applied, and where one ended and another began.

The models were described by one user as "...a reasonable way of conceptually partitioning the execution process". They found that the paper versions of the models left them rather confused, but that the execution description exercises of Dialogue 1 clarified

matters considerably, and they stated that the models were well-suited to the task of describing execution.

Some initial reservations were expressed by other users as to the number and relationship of the different viewpoints. Several users failed at first to grasp the difference between Search Space and Search Strategy, although all but one reported a later moment of illumination when engaged in Dialogues 2 and 3. One maintained that there were only two viewpoints in his opinion, Resolution and Search. This meant that he sometimes had difficulty in locating a suitable step description in the menus of Dialogue 1. A similar Search Space/Search Strategy difficulty was reported by another user who at first failed to distinguish "try goal with next clause" from "prove goal with Search Space". (This user later reported a 'Eureka' moment during a Dialogue 3 session).

A different conceptual difficulty arose for another user who suggested that Resolution should be presented as "... a kind of subroutine used by Search Strategy". This was the same user who wanted the changes from one model to another to be more clearly flagged in Dialogue 1, and their main difficulty seems to be in grasping the relationships *between* the models. This user also suggested that Resolution and Search Strategy models are more "algorithmic" in nature than the Search Space model.

Question 3.2.

- Please comment on the way that the system used the different rule-parts; (ie. to describe execution, to take input from the user, and to describe the effects of bugs).

This question generally evoked such responses as "fine", "quite natural", and in relation to Dialogue 1, "very clear". In relation to the description of bugged execution the use of the model-parts was described as "helpful" by one user, but "not always entirely clear" by another. A third user found himself "slightly baffled" by these descriptions. (These last two users appear to have forgotten the 'only one difference' condition under which the bugs operate).

Other users quibbled with the 'grain size' of the model-parts. One, while finding the use of the model-parts "fairly effective", found some confusion with the steps which ended the application of the Resolution model. After the last pair of arguments has been unified, users such as this one frequently wanted to move on to apply the Search Strategy model. A further Resolution step is necessary first, however, which states that *all* the argument pairs have been unified.

A similar difficulty was experienced by another user in relation to the need to state that *all* subgoals had succeeded after the proof of the *last* subgoal had been successfully completed. The user felt that this was rather too detailed an approach, but accepted it as a "...good rigorous approach to understanding Prolog execution", and said that they soon got used to it.

Question 3.3.

- Please comment on the way the system related different viewpoints to different categories of bugs.

The majority of users found the relationship between bugs and viewpoints clear and useful, although those who had earlier had some difficulty in grasping the execution models experienced some confusion. One user, initially confused, stated that "the more I used it the clearer it became". Another liked the bug/viewpoint relationship because "... the differences were made clear and thus you can see which viewpoint the bug lies in". A third user praised the strategy as a suitable way to implement a tutorial programme.

The same users who had expressed difficulty with the Search Space/Search Strategy distinctions in earlier responses did so again here. One user suggested that bugs such as the 'wrong clause order' bugs could equally well be analysed as a 'missing clause' bug, thus clearly showing that they had failed to grasp the implications of the 'only one difference' condition on the analysis of bugs. Similar points were made by another user,

although later reflection allowed them to clearly grasp the Search Space/Search Strategy bug distinction and to report this.

Question 3.4.

- Please comment on the exercise which asked you to identify the bugged clause, and the information available to you at this point.

The users generally liked the exercise of finding the bugged clause, with one stating that "... the problems were simple but good examples for me". Two users explicitly stated that all the information they required was readily available to them and easy to understand.

One user pointed out an unresolved conceptual difficulty in this exercise. This related to the bug 'wrong clause order'. If the 'bugged' database clauses appear in the order 'A, B, C, D', and the 'ideal' database clauses in the order 'A, D, B, C', which clause should be labelled as the 'bugged' one? Is 'B' wrong because it is where 'D' should be, or is 'D' wrong because it is in the wrong position? VIPER will, at present, insist that 'B' is the bugged clause, as it is the first 'bugged' database clause to show a difference in the execution histories to the 'ideal' database clauses. A possible solution for this problem is discussed in the section on further work.

One of the more experienced users had some difficulty with the exercise of locating the bugged clause as they could not initially suspend their knowledge of backtracking on one problem. Even doing so, they found that it was harder to determine what the correct code would be for Search Strategy problems, and suggested that more information on the Search Strategy viewpoint and the nature of Search Strategy bugs should be provided.

One user again found difficulty due to a lack of appreciation of the 'only one difference' condition. This lead them to state that a buggy program could always be fixed with a new first clause which gave the desired result.

Question 3.5.

- Were the exercises of Part 2, (describing the execution of the bugged code and identifying the bug), useful in relating viewpoints to categories of bugs? (Please explain).

The exercises of finding the bugged clause and of describing the execution of a goal with the bugged clause appeared to be quite powerful catalysts of the user's understanding of the relationship between viewpoints and bugs. A user with a few months intermittent experience of Prolog, ("more off than on") who had reported difficulties in distinguishing issues of Search Space from those of Search Strategy found that this exercise made them reflect on the bugs. They realised that Search Strategy related to the clause *order* while Search Space related to database *content*, "...then the light clicked, Eureka!". The dialogue is designed to promote just this sort of learning by focussing the user's attention on the exact effects of the bug. The 'Search Space' problem was shared by a second user, who felt that the issues could have become clearer if they had spent more time on the exercise. A third user found the exercises useful in clarifying matters once they had grasped the implications of the 'only one difference' condition.

The efficacy of these exercises were praised by a user who was aware that a given result could have been caused by a number of bugs. "The system made it clear which was the real bug. This helped in understanding which viewpoint it comes from". This user realised that the other bug could well have been associated with a different viewpoint, and felt that it was very useful to compare the two possibilities while using the system to find the actual bug.

One user with only a few weeks experience of Prolog expressed some reservations. They reported that running through the execution of the bugged clause sometimes made it clear how the bugged behaviour was occurring but sometimes did not. The failures occurred when the user's concentration on what the bugged clause *was* doing caused them to forget

what it *should* have been doing. They also felt that, having selected a particular viewpoint to work with, they were looking for that type of bug and ignoring the others. They suggested that a useful additional exercise would be to have the users find a bug by regarding the code from *all* viewpoints rather than selecting one initially. This indicates to us that the design of the dialogue made its purpose clear, and that it was effective in promoting learning by the user.

8.3.4 Responses to Questionnaire section 4, VIPER overall.

Question 4.1.

> • The system you have just used does not yet have any diagnostic mechanisms built into it, and can thus only adapt to a user in very limited ways. Please assume that such mechanisms could be added, and comment on the usefulness of the resulting system in relation to Prolog novices.

VIPER was generally well-received, with two users judging it to be already useful in its present form. All who expressed an opinion, (five out of seven), judged that the system augmented with diagnostic mechanisms would be useful or very useful.

Several justifications were advanced for these opinions. One was that the system allowed novices to work in their own way, "...finding the bugs using their particular method". Another was that VIPER already tutors the execution process well, and "...allows the user to diagnose a range of bugs that could arise from anything from a typo to a serious misconception". The role of diagnostic mechanisms was emphasised by a user who pointed out that often beginners (including himself) did not understand what was missing from their knowledge. Diagnostic mechanisms would help to identify and thus rectify such gaps. Another user made the point that an augmented VIPER could be very useful, as it would address a question "typical" of novices: "why did my code not produce the expected result?"

There were also some caveats. The lack of backtracking was seen as a drawback by one user, on the grounds that novices might get confused when they returned to writing real programs. It was suggested by a different user that real novices would need much more time to practice the exercises of Dialogue 1 to ensure that they had clearly grasped the three models before being "allowed" to go on to Dialogues 2 and 3. This user drew on their own experience to state that the exercises of the later dialogues would then "...not only help debugging skills, but clarify what had been learned in part 1".

One enthusiastic user provided detailed notes on how the current VIPER could be augmented with diagnostic mechanisms and natural language generation techniques. The first of these pointed out that the explanation/correction mechanism of Dialogue 1 does not take the actual choice that the user has made into account, but simply states the model-part that should have been applied. A more profitable strategy would be to compare the user's actual choice with the correct choice. This would allow a useful range of misconceptions to be detected and "repaired".

Examples were given of where this strategy could be applied. A choice of 'functors fail' when the correct answer is 'functors ok' could indicate that the use does not know what a functor is. Alternatively, the choice of an option from Search Strategy in the middle of a Resolution sequence could indicate that the user is baffled. The system could then offer to explain the viewpoints to the user.

Other suggestions were made for Dialogue 2. Here, the choice of a particular clause as being the bugged one is currently answered essentially by a 'yes' or a 'no'. The user acknowledged that the correction of misconceptions here is a more complicated matter, but suggested that the choice of bugged clause could be evaluated. The following was given as an example of what response the system could make if clause 1 was chosen erroneously:

> "No, clause 1 would need a new functor and an additional argument for the resulting program to give the correct result".

It was also suggested that when the user did choose the correct clause, the system could give some explanation of why this was the correct choice, (in case the user was guessing), or ask the user to give an explanation and critique it. The latter option was seen as more problematic. VIPER's current bug explanation facility is seen as being quite reasonable in relation to the former option, "...although a perfectly crafted piece of NL text could do the job better in some circumstances".

In Dialogue 3, the canned explanations of why an answer to a particular slot are incompatible with the answer given to the previous slot are judged to be "fine". The suggestion for this dialogue was that the user's answers should be compared with the correct answers and an explanation given in relation to any discrepancies. The following example was given:

> "That's basically right, except that the bug is 'wrong functor' rather than 'wrong arity'. If we change the functor in "big(X, Y)" to form "huge(X, Y)", then this clause unifies with the goal, producing the desired result".

These suggestions will be discussed below.

Question 4.2.

- The system can focus its activity on a particular viewpoint with which a student is having difficulty or which interests them, (eg. Search Strategy or Resolution). Please comment on the usefulness of this feature.

The ability to concentrate on a particular viewpoint was generally described by such terms as "excellent", "essential", "very good", and "nice". The reasons for these judgements were quite varied. One user who had only limited experience of Prolog found the

Resolution bugs easy to detect as he had written a lot of theorem proving and unification code in LISP, but found that the Search Strategy and Search Space viewpoints were new to him. VIPER's choice mechanism allowed him to focus on these. Another user compared the viewpoints to hypermedia in that they allowed a student to focus on sub-problems or sub-issues rather than deal at all times with a complete problem space. This was seen as a useful feature. One user had particular problems with Resolution, and was very pleased to be able to concentrate on this viewpoint with considerable success.

Once again, reservations were expressed by some users. One expressed the fear that students might simply continue to work with the viewpoint that they were best at, and suggested that VIPER should be able to shift the focus of the work on to another viewpoint should it become necessary. A different user made the point that although the choice feature was very useful, a novice might not be able to understand clearly which viewpoint they were having difficulty with.

Another response touched upon a more profound issue. This user stated that the final goal was to become good at debugging, and that the learning of viewpoints was an excellent way of achieving this. Their reservation concerned the way that VIPER's exercises promoted learning in a particular direction, that is from viewpoint to bug; (ie. the user chooses a viewpoint and the system offers a bug to be solved by using that viewpoint). The suggestion was that to promote realistic debugging, learning should also be encouraged in the opposite direction. The question in this case would be, "given a bug, how do I find a viewpoint that will help me to solve it?". Tutoring in these terms was seen as an essential complement to the exercise that VIPER currently conducts. (If VIPER had been developed to include an explicit statement of the area of application of each viewpoint, as suggested in chapter 3, then such tutoring would have been simple to accomplish). Also, it was stated that during the 'viewpoint to bug' exercise, it should be made "very clear" that users were *not* starting from a bug and looking for a viewpoint, since that is what most of them would expect to be doing.

This same user made some other pertinent points. They stated that "upward compatibility", or an extension of the viewpoints to include the full functionality of Prolog, would be highly desirable. One reason for this was that it would give VIPER more flexibility to cater to users who wished to learn in the 'bug to viewpoint' direction described above. Another reason was that it would facilitate a new exercise where the student was able to alter the bugged code to determined whether or not the alteration produced the desired result. For this exercise, the student would be allowed to make *one* change *within* the current viewpoint only. The assumption behind this was that even within one viewpoint, there may be more than one way of solving a bug, and that having a single ideal version of the code was "...a very artificial contrivance which is not upwardly compatible". While the use of an 'ideal code' structure may be suitable for novices, the user stated that it was very hard for those who already had a better knowledge of Prolog to confine themselves to a single change in relation to one piece of ideal code.

Question 4.3.

- Did working with the system add anything to your understanding of Prolog execution? Please explain.

Two users reported that their time with the system had added little to their understanding of Prolog, as they had already had considerable experience with the language. However, one of these gave the opinion that "...for novices it should be excellent".

The other users reported a variety of learning outcomes which were frequently related to the description of execution in terms of different viewpoints. Learning for one user involved an appreciation of the fact that execution can be described in terms of different categories, and for another, a better appreciation of what those categories were. A third user reported that he had always viewed Prolog in terms of Search, but that he

"...had never considered breaking it down into the three viewpoints to identify bugs quickly or follow the execution from those viewpoints".

The fine-grained detail of execution was a revelation for one user who had never considered Prolog at this level before. The user who had written theorem-provers in LISP reported that he had learned something about how Prolog goes about proving goals, and stated that for beginners, the system would probably be very informative.

8.3.5 Responses to Questionnaire section 5, the Viewpoints.

Respondents were asked to answer these questions **without** reference to the specific system they had just used

Question 5.1.

- Are these viewpoints on Prolog useful? (Please explain)

All users found the viewpoints useful in one way or another. One report stated that they broke Prolog execution down into manageable chunks, and another that they would help that user to debug in future. They were described as "...useful conceptual clarifications on the way Prolog works" by a user who, as reported above, insisted on the need for "upward compatibility" in developing the viewpoints. The viewpoints were useful "...both for tutoring and for finding bugs..." according to a different user, who maintained that "...this kind of conceptual separation is clearly necessary in viewing a complex task like Prolog debugging". This user was satisfied with the clarity of the viewpoints, although they still had some doubt about the distinction between Search Space and Search Strategy.

This doubt was shared by two others. One of these expressed reservations as to the usefulness of the Search Space viewpoint, but tempered this by saying that they already thought of Prolog in terms of Search, and so probably took the Search Space viewpoint as given. They went on to say that novices would probably find it more useful.

Question 5.2.

- Can you think of any other viewpoints which would be useful?

Only two users suggested other possible viewpoints. One reported that they viewed 'Resolution' more in terms of 'unification', ie. they viewed two resolving expressions as 'tree' structures and then"... unified those in the normal way". They suggested that this might be a useful alternative viewpoint for students such as computer scientists, but stated that the one presented by VIPER is good for novices.

The second user suggested a viewpoint "...that reflects the 'state' of all the current unifiers...", but qualified this by saying that it might be too confusing for novices. This suggestion is discussed below. A further qualification was the suggestion that viewpoints should be totally "disjoint".

8.4 Discussion.

8.4.1 Discussion of responses to Questionnaire Section 1.

This section showed that the users who took part in the evaluation had a spread of ability from 'poor' to 'good', and that this ability did not depend simply on the length of time for which the individual user had been learning Prolog. Although a majority of users stated that they had a clear model of Prolog execution before the tutorial with VIPER, this did not imply that they could learn nothing from the system, as one user who rated his ability as 'good' immediately stated that their understanding of execution was clearer as a result of the tutorial.

8.4.2 Discussion of responses to Questionnaire Section 2.

The responses in this section indicate that the interface was generally good at facilitating dialogue between system and user, and certainly did not seriously impede the progress of the tutorial. A number of problems were reported, some being related to the LPA Prolog environment, and some to the size of the Macintosh screen. A number of suggestions were made for improving the interface.

The problems of the scrolling menu that returned to its 'top' position, and the erroneous variable identifiers which caused the meta-interpreter to fail, could possibly be cured after some consultation with the Prolog environment's designers. The problem of two 'bug' examples looking identical could be cured by making the examples more distinct.

The users who requested wider menus could be satisfied where the menu was of the 'popup' variety. Where the menus are constantly displayed as a part of a larger integrated screen, extra code would have to be added to the system to ensure that long lines of displayed code were fitted into the menu's present width.

Only limited responses are possible to the complaint that explanation sometimes overlaid the tutorial display, unless the system is run on a machine with a larger screen, (eg. a Macintosh II). As suggested in a response to question 2.4, the user could be allowed to switch between explanation and tutorial screen. Such switching is automatically possible in the LPA environment, but a specific mechanism would have to be installed to make this functionality clear to a naive VIPER user. The problems reported by one user in moving from pulldown to hierarchical menus are an unavoidable feature of the environment that VIPER is built in.

The user who complained about the number of mouseclicks required in Dialogue 3 could be satisfied by programming the relevant menus to popup automatically after input had been

made to the preceding slot. However, there is possibly some pedagogical value in requiring the user to positively select each button in turn before giving input to a particular slot. The value could be that the extra button clicks help to focus the user's attention on to the aspect of execution that is being described, and that they help the user to learn and remember the order and relationship of the different parts of the description. A compromise solution could involve the use of a diagnostic mechanism. In this case VIPER would only present the popup menus automatically if the diagnostic mechanism indicated that the user had previously understood the structure and relationships of the Dialogue 3 execution description.

The responses to question 2.3 were very satisfactory. The structures of the 'Questions' menu and the explanation facility of Dialogues 2 and 3 were based on the 'model and operators' formulation of a viewpoint given in chapters 3 and 4, and whose implementation is described in chapter 7. The praise given to these parts of VIPER is taken as an indication that this aspect of the design based on models and operators is relevant to users' needs, and adequately implemented.

The additions suggested in responses to section 2.4 could usefully be implemented in VIPER. The additions to Dialogue 1 which would help a user to recall the overall state of the execution, and which would highlight the transitions from applying one model to applying another, appear to be highly pertinent and directly related to VIPER's pedagogical goals. If successful, the former would reduce a user's confusion, while the latter would make the relationship between the models more explicit.

The suggestions for Dialogue 2 were equally apposite. A statement of the viewpoint being exercised and the bugs it might involve would most likely help to structure a student's thinking, and support their learning of the relationship between viewpoints and bugs.

An equally positive response can be given to the suggestions for Dialogue 3. The intention that VIPER should give some assessment of a user's input to this dialogue is implicit in the

system design, and is explored further below. The predicates which control this Dialogue could well be adapted to allow a user to change previous input. In order to maintain the logical coherence of the input after such a change, the entries for all subsequent slots would have to be checked in turn, or new input requested.

8.4.3 Discussion of responses to Questionnaire Section 3.

The exercise of applying the three models to describe the execution of specified code seems to have been very successful as a method of establishing a knowledge of these models as a pre-requisite for Dialogues 2 and 3. Confusions that had been created by the paper versions of the models were, in general, quickly dispelled. The models were praised as a "rigorous" description of execution which was suitable for beginners, although the relationship between the models could have been presented more clearly. The relationship of the Search Space model to the others gave particular problems at this stage, and it is clear that some users initially failed to appreciate the implications or relevance of this model. All but one of these found their understanding greatly improved by the 'debugging' exercises of Dialogues 2 and 3, and it is thus concluded that this model is relevant and useful. Dialogue 1 could be improved by making the relationships between models more explicit.

The responses to question 3.2 indicated that the models of section 6.5 were generally well-suited to their use in the system. Their use in explanations of the bugged execution could have been clarified by a re-statement of the conditions under which the bugs were defined. It is possible that the models could be improved by removing two parts which were judged to be somewhat too detailed. These are the statement that all arguments of a predicate and goal have been unified, and the statement that all subgoals of a clause have been proven. This issue is possibly best decided by further empirical work.

The relationship between bugs and viewpoints was explored in question 3.3. Here again, the formulation seems to have been successful and relevant. The formulation was praised as being suitable to a tutorial program. Even with the short tutorial time available, all the

users seemed to establish a clear understanding of the relationships involved. All but one of those who had failed to appreciate the significance of the Search Space viewpoint stated that their understanding was greatly improved by the exercises of Dialogues 2 and 3. It is significant that the exercise of bugfinding clarified the users' understanding of the viewpoints, as these viewpoints were in fact constructed so as to support the localisation of bugs. This is taken as an indication that each viewpoint is indeed applicable to the localisation of a particular range of bugs, at least in the limited world that has been implemented, and that this quality could be tutored explicitly. In this context, a statement of each viewpoint's applicability would be seen as an example of the 'application' heuristics described as a component of a viewpoint in chapter 3.

The success of the Dialogues 2 and 3 in establishing a relationship between viewpoints and bugs is clearly shown in the responses to question 3.5, where those who had failed to understand the Search Space viewpoint again reported a significant clarification as a result of their work in these dialogues. We take this as an indication that both the formulation of the domain and the design of the dialogues are effective in promoting the desired learning.

The effectiveness of the viewpoint/bug mapping is shown rather perversely by the user who complained that choosing a particular viewpoint for the exercise led them to ignore all the possible bugs that might be related to other viewpoints. If this is to be regarded as a problem, then it indicates that the viewpoint/bug mapping is very strong. VIPER does not seem to achieve this by unduly constricting the user's behaviour. One user reported that while the system made it clear which bug from which viewpoint he was studying, it also allowed him to compare this with another hypothesised bug from a different viewpoint. The suggestion of an additional exercise where the user had to find a bug by considering all three viewpoints rather than the chosen one indicates that the system and dialogue design have a clear logic that is appreciated by these users, and which implies for them that some useful extensions could be made to VIPER. The modularity of VIPER's design and implementation means that this extension would be very easy to implement.

As with the explanations of Dialogues 2 and 3, some users had forgotten the conditions under which the bugs were defined, and again showed some confusion regarding the viewpoint/bug relationship as a result. A re-statement of this relationship during the Dialogues might help to rectify this.

Not surprisingly, the bugfinding exercises of Dialogue 2 seem better suited to novices than to more experienced users of Prolog. The experienced users found some difficulty in suspending their knowledge of backtracking, and in thinking about bugs as being only a single difference from an ideal version of the code. The less experienced users found the exercises more helpful in clarifying their knowledge of the relationships between viewpoints and bugs. The access to information through the 'Questions' menu was appreciated as providing the data that was required in a form that was easy to understand. This mechanism also allowed the students to explore the problem in their own manner.

The conceptual problems relating to the 'wrong clause order' bug will have to be resolved, and VIPER's presentation structures amended accordingly. Some users may require more support in dealing with the Search Strategy viewpoint, and Search Strategy bugs, as one user reported a difficulty in determining what the correct code would be in this category.

A criticism of Dialogue 3 will also have to be addressed. This stated that the user's concentration on what the bugged clause was actually doing led them to forget what it should have been doing. This seems reasonable, and may require an augmentation of the Dialogue to focus attention on the desired behaviour.

8.4.4 Discussion of responses to Questionnaire Section 4.

These responses were very encouraging. VIPER was judged to be useful even as it stood by two of the users, and several others clearly saw a positive role for diagnostic mechanisms. Diagnosis, and adaptive tutoring based on it, were seen as being especially

important for novices who did not understand what was missing in their knowledge. Other responses indicated how such diagnosis might be achieved using VIPER's mechanisms, (these are discussed below). VIPER was judged to deal with a domain that was highly appropriate to novices, and to describe execution in a way that would make it very clear to them.

In order to answer some reservations, VIPER's domain would have to be extended to include backtracking, and the diagnostic mechanisms would have to ensure that the students had spent long enough on Dialogue 1 before they progressed to Dialogues 2 and 3. Both of these developments could be achieved without great difficulty.

The suggestions that were made about the siting and function of diagnostic and natural language mechanisms are particularly welcome, as we wish to make some claims about the diagnostic possibilities, and these suggestions frequently mirror the thinking of VIPER's designer. The simplest case is that of Dialogue 1, where a chosen step can be compared to the correct step in order to detect any misconceptions or confusions which the user may be harbouring. This technique, of using a 'bug catalogue" to recognise misconceptions and faults in a user's knowledge, has been widely used in previous systems such as DEBUGGY (Burton 1982), and while its potential may be limited, it has at least been carefully researched.

The suggestions for Dialogue 2 are more complex. The first of these involves explaining why a particular choice of bugged clause is wrong. The example given in the response appears to assume that there is only one way that the chosen clause could be changed to produce the desired result, and it is not clear that this is in fact the case. A simpler possibility with the current system would be to ask the user to specify which aspect of the chosen clause they thought was defective, and have an explanation mechanism which compared the detailed aspect with that of the ideal code. Erroneous answers at this stage could also be the subject of diagnosis.

The second suggestion for Dialogue 2, that the student should justify their choice with an explanation of the bug which is then critiqued by VIPER, is acknowledged to be more problematic. It is to deal with this issue that Dialogue 3 was designed.

The suggestions for Dialogue 3 are whole-heartedly supported. These recommend that explanation should be given in relation to any discrepancies between the user's input and the system's own description. The only point at issue here is the degree of complexity and sophistication that the diagnostic mechanisms which compute the explanation should seek to achieve. The implementation of these various suggestions for extending Dialogue 3 is discussed in the section on further work.

The second question in section 4 enquired about the usefulness of being able to concentrate on a single viewpoint. This facility is seen as one of the major advantages of a system incorporating multiple viewpoints, and the very positive assessments of it are gratifying. The users did concentrate on the viewpoint that most interested them, and appreciated the way that the viewpoints broke the 'problem space' down into smaller chunks.

The suggestion that a complementary tutorial should offer a bug and ask a student to identify an appropriate viewpoint is intriguing. It is indeed a logical development from VIPER's present exercises, and reflects the overall pedagogical goals of the system. Once a student has learned what bugs may be associated with each individual viewpoint, we would want them to apply this knowledge in a less clearly defined or 'protected' situation of the kind suggested. This accords with our emphasis on the *application* of knowledge in chapter 3, and may provide an opportunity to use the 'generalisation' strategy of Cognitive Apprenticeship listed in section 3.3.2. The development would be simple to implement and is discussed in the section covering further work.

Two other suggestions also inspire agreement. The first of these was that VIPER's viewpoints should be upwardly compatible with more elaborate ones which incorporate the

full functionality of Prolog. This is quite possible. The idea was briefly discussed in section 6.2.5, and is covered again in the section dealing with further work. The second suggestion was that users be allowed to make a single change to the code which was within the current viewpoint, to determine whether this produced the required result. Acceptable changes would have to be limited to those which would rectify a specific bug. Given the ready availability of the meta-interpreter this seems like a very sensible suggestion, and has also been made by others[1]. The altered code could be run through the meta-interpreter with the original goal to determine the result that it gave, and suitable feedback given to the student. This could produce a system with some similarities to SOPHIE, where a student was able to propose new measurements which were to be made under specific conditions in an electronic circuit. The restrictions placed on the changes that could be made would prevent the code from being changed to something that exceeded VIPER's analysis capabilities.

The final question in this section asked whether their tutorial with VIPER had added to their knowledge of Prolog. Little may have been expected here, given the users' considerable experience of the language and the short time that they used the system. A quite respectable degree of learning was, however, reported. For several users, this concerned either the notion that execution could be described in terms of different viewpoints, or else the exact nature and relationship of those viewpoints. A clarified perception of the relationship between Search Space and Search Strategy was reported by some, while an improved appreciation of how Prolog proves goals was reported by another. Even the users who reported no learning expressed the opinion that the system would be very informative for novices. On a general level, these responses are taken as implying that the research direction pursued through the implementation of VIPER does have real value.

---

[1] du Boulay, B., personal communication.

8.4.5 Discussion of responses to Questionnaire Section 5.

These responses were consistent with those given in section 4. The viewpoints used were judged to be useful conceptualisations of Prolog execution which were essential for the tutoring of both execution and debugging to novices. The reservations expressed are not deemed to be significant.

Two other viewpoints were suggested which might well be useful for more specialised users. The description of Resolution in terms of the unification of 'trees' seems to entail more that a simple substitution of representations, as it situates the process described in a wider and possibly more familiar context. This view may well be of use to computer scientists approaching Prolog, as the user suggested.

Another suggestion was for a view which reflected the current state of all the unifiers. This seems to have much in common with the 'Slices' of Weiser and (1986) and thus could have a special relevance to more advanced debugging.

8.5 Conclusions to Chapter 8.

Section 8.1 introduced the work which was intended to evaluate the implementation described in chapter 7. This implementation supports three kinds of dialogue which are intended to demonstrate the various mechanisms which are available in VIPER, and which, it is claimed, could be augmented to support adaptive tutoring by the use of well-known methods. The evaluation is intended to demonstrate that the mechanisms as they stand can support a tutoring interaction that is deemed to be useful or effective by the human users who engage in it.

Section 8.2 detailed the method and materials to be employed in the evaluation. VIPER was configured to provide a tutorial based on the three dialogues. The exact progress of

this tutorial depended on choices made by the user. A variety of printed materials were used to introduce users to the evaluation, and to support their progress through it. At the end of the tutorial, the users were asked to complete a questionnaire which enquired into their knowledge of Prolog, their experience of the interface, their views on the formalisms used in the system and the overall system design, and their views on Prolog viewpoints in general.

Section 8.3 reported the results of this questionnaire, which were generally very favourable. Some improvements to the interface were suggested, and the formalisms used by the system were judged to be well-suited to the needs of novices. This was especially true of the description of execution which was judged to be rigorous and clear. In spite of some reservations expressed about the role of the Search Space viewpoint, the responses demonstrated that real learning had been achieved even with the system as it stood. This frequently involved the association of specific bugs with specific viewpoints. The strength of this mapping indicated that the formalisms developed in chapter 6 are well-suited to the tutoring of bug localisation, at least in the limited domain that has been implemented. A number of specific proposals were made for the augmentation of the system with diagnostic mechanisms.

These results were discussed in detail in section 8.4, and conclusions drawn as to their significance and relevance to future development of the system. It is concluded that the research direction is a fruitful one, and that much could be gained by augmenting VIPER with diagnostic mechanisms whose potential and design have been demonstrated with other systems. Other possible developments, such as those which would allow the user to make limited changes to the code are also discussed.

Chapter 9. A Discussion of VIPER.

Introduction.

This chapter considers the issues raised in VIPER's design and implementation. Some are issues which motivated the project, and others are issues which emerged from its execution.

These issues are discussed under the following headings:

- The viewpoint formalisation.
- The design goals of VIPER.
- VIPER's design as a realisation of these goals.
- Cognitive Apprenticeship.
- VIPER and other domains.
- Future Work.

9.1 The viewpoint formalisation.

This section relates the method of formalising viewpoints which is presented in chapters 3 and 5 and used for the implementation of chapter 7 to some of the issues raised in the thesis. A number of major points about viewpoint structure are made, followed by some less significant ones in relation to tutoring system design.

The discussion of viewpoints in chapter 2 identified two general roles for multiple viewpoints in tutoring systems. The first role was the use of multiple viewpoints on a domain to support a single activity in that domain, as where SOPHIE II (Brown et al. 1976) uses quantitative and qualitative views to support the debugging of electronic circuits. This is the role most relevant to VIPER, where different viewpoints are used to support the single activity of debugging in a simplified domain. We would thus claim to have successfully implemented a system to utilise multiple viewpoints in this manner.

The second role identified in chapter 2 was the use of different viewpoints to support different activities, as with STEAMER (Hollan et al. 1984). The essential point in this case was that the way an individual perceived the system was dependent on the activities they wished to carry out with it. As VIPER is only aimed at the single activity of debugging, this role is not seen as relevant, and is not discussed any further.

The final structure for formalising viewpoints which is presented in chapter 5 describes three classes of operators which are applied to a model to produce inferences. The implementation described in chapter 7 uses only the first two of these, the simple "access" operator, and the "inference operator" which links two separate parts of the model to make an inference. The third class of operator is designed to transform the model by adding or deleting information. This class of operator was omitted in order to keep the implementation project within the relevant constraints of time and space. It is envisaged that this operator could well have a role in future work, where it could be used to simulate misconceptions in relation to the various viewpoints on Prolog execution, or else to simulate a user's progress from elementary to more advanced viewpoints.

Two other points emerge from the discussion of chapter 3. The first is that, to enable adaptive tutoring, the viewpoints, or the process that chooses them, should encode information about the relevant goals and learning histories of the students. The very limited student modeling capabilities built into VIPER mean that this facility has not been implemented, but is discussed further in terms of future work.

It was also suggested in chapter 3 that the viewpoint formalisation should be modular, so as to facilitate the diagnosis of student errors and misconceptions. The viewpoints described in chapters 6 and 7 are in fact highly modular, but the advantages of this for diagnosis have not been demonstrated, as no diagnostic capabilities have yet been added to the system. The possibilities of doing so are discussed in the section on future work. The evaluation discussed in chapter 8 does indicate a different benefit of the modularisation, as

it shows that the viewpoints do help the users to divide up the 'problem space' of debugging in to smaller, more manageable, parts.

A number of less specific, or less significant points can also be made about the viewpoint formalisation. The first of these relates to the discussion in chapter 3 of Cognitive Apprenticeship, and the points raised by Brown, Collins and Duguid (1989). Brown et al.'s analysis sets the goal that the viewpoints used by a system should support the successful execution and explanation of tasks in the relevant domain, in order for tutoring to be embedded in ongoing practice. We would claim success in relation to this goal. VIPER can demonstrate and critique descriptions of code execution, and can localise and explain bugs in a simplified domain by using the viewpoints that are tutored.

Another general point arises from the discussion of chapter 3. This indicated that the formalisation of a viewpoint should encode information about the viewpoint's area of application. This information should make clear the connection between seeing a problem in a different way, and solving it. This encoding has not been carried out explicitly in VIPER, but is implicit in the mapping from the models onto the bugs. This point is discussed further in the section on VIPER's design. An explicit statement of the area of application of a viewpoint would allow a specific strategy to be stated for choosing a viewpoint to solve a problem, as suggested by Stevens, Collins, and Goldin (1979), and would enable the tutoring and explanation of the relationship between viewpoints.

In chapter 2 the discussion of the previous use of viewpoints in ITS concluded that such viewpoints could be characterised in two ways. Firstly, they were seen as complementary modes of analysis, both of which might be required for the solution of a given problem. Secondly, they referred to the same set of objects, but identified and structured them in different ways. VIPER's viewpoints are only complementary when they are used to describe code execution. They have been explicitly formulated so that when they are used for debugging in the simplified domain, it is not necessary to use the viewpoints *in*

combination to characterise the bugs. The second characterisation is more relevant. The operators which provide explanations of the bugs in terms of the different viewpoints do analyse the code in different terms, and do observe and respond to different features in the execution histories. We can thus claim a consistent account of viewpoints, from the identification of their significant features, to their formalisation and implementation.

9.2 Design goals of VIPER.

This section considers a number of goals for VIPER's design which are mainly articulated in chapter 3. These goals specify how the use of multiple viewpoints should be related to tutoring, explanation and demonstration in the chosen domain.

The goals stated in section 3.2 indicated that the implemented system should be able to tutor using two or more viewpoints on a given domain.

In operational terms this meant that the system should be able to:

a) tutor viewpoints independently;

b) make clear the area of application for each one;

c) tutor the relationship between viewpoints and their use in combination.

The first of these goals, the ability to tutor the viewpoints independently, has been achieved in relation to the debugging exercises by allowing the student to choose the viewpoint they wish to work on and providing mechanisms which tutor in relation to it. (It is not claimed that these mechanisms are intelligent). The pedagogical value of this is that the large problems of debugging, and learning about debugging, are reduced to manageable 'chunks' which can be tackled individually. This facility is thus a central feature of VIPER's design. As some of the responses to the evaluation suggest, the next development step could well be to put all the viewpoints 'back together', having learned

them individually, so that the student has to find a bug without knowing which viewpoint it is related to.

The second goal, that of making clear the area of application for each viewpoint, has been achieved, although the only explicit statement of the relationship between viewpoints and bugs was in the printed materials given to the evaluation participants. The conventions which map the viewpoints onto the bug catalogue (see chapter 6), and the exercises which identify the bug, implicitly make clear the kinds of bugs which can be localised with each viewpoint, (as they were designed to do). An example of this can be seen in the evaluation responses where those users who could not see the point of the Search Space viewpoint suddenly understood that it related to missing or extra clauses. Thus the lack, in the system, of an explicit statement of each viewpoint's area of application did not seem to impede those students who took part in the evaluation. Were such a statement to be provided, however, it might well be useful in the correction of misunderstandings in relation to the use of a particular viewpoint. This would be particularly relevant if students were asked to find a bug without knowing which viewpoint it was related to, and attempted to apply the wrong viewpoint.

The third goal, that of tutoring the relationship between viewpoints, and their use in combination, was not attempted in VIPER.

Section 3.2 also sets the goal that the system must be able to provide explanation in relation to each viewpoint. This was achieved in the execution description exercise by providing explanations structured around the different parts of each model, and by including a facility which could demonstrate the application of the model parts to the current execution step. Explanations in the bugfinding exercises were structured around the differences between the bugged and ideal code, and the patterns of resolution success and failure in the execution. The predicates that produce these explanations are examples of the "inference operators" described in chapter 5. Once identified, the mapping conventions related the

different bugs to the different viewpoints. The inferences and observations stated in the explanations could be replicated by the student using the 'Questions' menu.

The desire for 'glass box' representation of domain, and the demands of Cognitive Apprenticeship for demonstrations of expertise in the domain, required that the system should be able to execute the domain tasks itself using the relevant viewpoints. For the exercise of describing code execution, this was achieved by repeatedly using the predicates which applied the correct model part to the current execution step. For the bugfinding exercises, the explanations are themselves demonstrations of expertise in the domain, as each uses an explicit inference procedure to discriminate between the various possible bugs.

9.3 VIPER's design.

The focus of this section is the system implementation described in chapter 7. The issues raised relate to VIPER and debugging, black-box verses glass-box solutions, explanation, the dialogue structures and the system architecture.

9.3.1 VIPER and Debugging.

The viewpoint formalisation given in chapter 3 was intended to:

a) allow the tutoring system to function successfully in the relevant domain;

b) give demonstration and explanation in relation to each aspect of the domain;

c) set tasks which relate directly to the real-world domain.

The first two of these goals have been realised, as described in section 9.1 above. The third has been realised in a limited sense. The nature of this limited realisation is now discussed in relation to the three procedural models of Prolog used in the system, and the simplified debugging domain in which they are applied.

The three models of section 6.5 which form a part of VIPER's domain specify a subset of Prolog's behaviour. Thus describing code execution using these models in VIPER is in principle no different from describing code execution in the 'real world' where such code might be written. The limitation on the achievement here is a result of the omission from the models of several crucial aspects of Prolog, such as backtracking and the use of the 'cut'. This deficiency could be addressed by extending the models to specify the full functionality of Prolog. Within this limitation the models seem to have been very successful, giving novice students a clear and rigorous means of describing and predicting execution.

The discussion becomes more complex in relation to the 'debugging' exercises supported by VIPER. As described in chapter 6, these exercises are carried out in a much-simplified domain. This strategy is seen as necessary both to support the activities of novices and to define a manageable domain. For novices the domain can be seen as analogous to the 'shallow end' of the swimming pool, while the use of an 'ideal' version of the code allows us to circumvent the intractable problems of building a real debugger; (see section 2.4). The assumption behind VIPER's exercises is thus that they are usefully related to the real world, and that any learning which takes place during an interaction with the system will transfer to that real world. This is not to claim that VIPER's activities are fully 'authentic' as Brown et al. (1989) would advocate, nor that they constitute an explicit 'theory of debugging' which is to be tutored. Neither claim can be made due to the artificial limitations and simplifications of the domain formulation.

The claim is that rather than having a theory of debugging explicitly represented, VIPER *embodies* such a theory. In this sense the use of different viewpoints on Prolog *is* a theory of debugging, even if VIPER does not support a real debugging environment. In other words, the viewpoints are authentic, (if incomplete), even if the 'debugging' domain is not, and it is to facilitate the learning of these viewpoints that VIPER has been designed. We would argue that the viewpoints have the domain transparency that Brown (1989)

advocates, since that they are learned *in order to localise bugs*, and not just for their own sakes. The most detailed assumption here is that their application in the simplified domain will transfer usefully to the real domain. This can only be tested empirically, although the indications from the first evaluation of chapter 8 are positive, with even experienced Prolog students having 'Eureka' moments in relation to the significance of particular viewpoints. The domain as it stands could be considerably enriched by upgrading it to include bugs which reflect the full functionality of Prolog, while still analysing these in relation to an 'ideal' version of the code.

At this point it is worth digressing briefly to consider what a more 'authentic' exercise in debugging might entail, assuming that the same meta-interpreter and execution-history architecture is used for the system representations. Generally it entails an inability to choose between different 'fixes' for a program, so that the system cannot test and critique a student's understanding of the effects of the original bug. This can be demonstrated by a hypothetical exercise where the student is allowed to make a single change to the bugged code, the change being limited to one of the 'approved' bugs. All that would have to be specified for this would be the desired result, so that there is no longer any need to have an 'ideal' version of the code. The changed code could be run through the meta-interpreter and the result reported. There are, however, always many ways of fixing a bugged piece of code, and the exercise just described gives no basis for choosing among them, or for critiquing a student's understanding of the bug. Since the symptom under examination in VIPER means that the query always succeeds ultimately, then there are many cases where simply re-ordering the clauses will produce the correct result. Alternatively, a new clause could always be inserted at the beginning of the database.

The exercise could be constrained by such means as prohibiting certain fixes, insisting that a fix related to a particular model should be used, or stating that a particular clause should be changed, but none of these solve the essential problem of inability to choose between alternative possible fixes. The result would be a rather superficial exercise where the

desired result was obtained without the student being helped to appreciate how each viewpoint could localise particular bugs. Also, if the student's change does not produce the desired answer, then the system has no means of explaining why, or demonstrating the 'correct' change, unless it assumes that one of the many possible changes is in fact the 'correct' one. Having the system generate all possible solutions would create a very large set of alternatives, and it is not clear how this would be tutorially useful. The point seems to be that for 'real-life' debugging, there are many other constraints beyond that of getting the 'right result' which determine the fix to be used, and which are artificially absent when a 'debugging exercise' is set.

An alternative approach could be to develop a 'theory of debugging' based on the model-parts of section 6.5, as these are an abstract description of code execution. This approach again assumes that an 'ideal' version of the code is used, but does not require the 'only one difference between bugged and ideal code' condition and could thus claim to be more authentic. The idea of 'inference operators' given in chapter 5 could be used to define sequences of model-parts from a single model, each of which would constitute a particular 'operator', and would describe some part of a given execution history. These operators make explicit information which is only implicit in the model, by the use of an inference procedure. The example given previously was: (A -> B, B -> C) -> (A -> C).

In the following examples, '->' should be read as 'implies', while 'trace evidence' should be read as indicating that evidence for the described condition will be found in the relevant execution history, thus allowing the operator to succeed. These examples are intended to be similar in structure to the explanations available in VIPER via the "Questions" menu. The provision of these explanations is described in chapter 7.

The following could be examples of such operators for the Search Strategy model:

1. 'Try next clause' with clause X -> success -> trace evidence.
2. 'Try next clause' with clause X -> fail -> trace evidence.

3. 'Try next clause' with clause X -> success -> subgoals succeed -> trace evidence.

4. 'Try next clause' with clause X -> success -> subgoals fail -> trace evidence.

5. 'Try next clause' with clause X -> success -> desired value present -> trace evidence.

(For brevity's sake the parts of these operators do not exactly match the form of the Search Strategy menu choices for describing execution).

A sequence of such 'operators' would form an abstract description of a particular execution history in terms of the specified model. (This would be different to the standard execution history description of VIPER's fist dialogue, as model parts relating to other models would be omitted). The task set for the student would be to identify the *expected* sequence of such operators which would produce the desired result, as opposed to the *actual* one which produces the bugged result. The system would know the desired operator sequence from its inspection of the execution history of the ideal code.

Such a scheme would have the advantage of relating the viewpoint models directly to the bugged behaviour, (although how a clear mapping is to be achieved is a question which would require further research), and of providing a formalised means by which the system could make inferences and provide explanation. It would also allow each bug to be described in terms of each model. However, the same problem arises as with the other alternative exercises. If the student's suggested sequence of operators does not exactly match that shown by the ideal code, how is the system to know that it may not, in fact, describe a perfectly legal path to the desired result? In addition, the need to learn about the structure and interaction of the operators represents a large cognitive overhead which would be of little immediate benefit to the student.

We conclude from this digression that while VIPER's domain and exercises may well be 'inauthentic' in some senses, it is not at all clear how a useful *tutorial* system could be built

which was more 'authentic' without being a full-blown debugger, assuming that such a thing can be built.

The earlier decision to use a simplified bug domain rather than to attempt to build a debugger seems to have been vindicated. The decision allowed us to concentrate on building mechanisms which described and tutored each bug's effect on execution (see chapter 6). The benefits of this seem to have been a clarity in the mapping from viewpoints to bugs which aids both the novice and experienced Prolog user. Not only does this mapping make the bugs which can be localised with each viewpoint very clear, but, as the evaluation responses in relation to the bugfinding exercises show, it promotes a more complete and accurate learning of the viewpoints themselves.

The simplified domain did seem well-suited to the needs of novices. Those who took part in the evaluation were quickly able to set about the bugfinding exercises, once they had understood the models and the 'ideal code with one difference' conditions.

9.3.2 Black boxes and Glass boxes.

The issue of black and glass boxes was discussed in section 2.1.4 with reference to WEST (Burton and Brown 1979), and it was noted that in some cases they could profitably be used in combination for tutoring where the tutored domain was not over large. As stated in section 3.1, we do not wish to claim that representations based on viewpoints have a 'psychological reality', but rather that they are useful performance simulations of reasoning in the domain. This discussion is relevant to VIPER as two different 'bugfinding' mechanisms are used by the system. As described in chapter 7, the 'bug-recognisers' exploit the structure of the domain to identify the specific version of the bug that is present as efficiently as possible, and make no pretence of being useful for explanation of this process. With the specific version of the bug identified, a text template describing its effect and nature can be composed. The 'bug-explainers' do not carry out such a detailed analysis, but identify the bug in sufficient detail to provide an explanation of it which is

structured around the 'inference operators' of chapter 5 and the viewpoint-to-bug mappings of chapter 6. It is this structure that gives the explanations their generality.

The bug-explainers are thus intended to be a 'glass-box' representation of the domain, and as such were certainly an improvement on the 'bug-recognisers'. Our intention was that, as they were structured around explicit and uniformly-structured inference procedures, the students would be able to explore and check this inference procedure themselves by using the 'Questions' menu, once an explanation had been delivered. It is not clear that this exploration occurred, and it has to be said that the reaction to the explanations was mixed. Some students found them very helpful, while others apparently failed to grasp the structure of the domain and only found the explanations confusing. We assume that with more prolonged exposure to the system the structure of the explanations would become more meaningful to those who found them initially confusing.

The presence of the two bug-identifying mechanisms in VIPER does constitute a degree of redundancy, although they support quite different functions. This could be avoided by developing the 'bug-explainers' to be as powerful at discriminating versions of bugs as the 'bug-recognisers'. Alternatively, the 'bug-recognisers' could be given a new role in an author interface for VIPER. Since they efficiently identify all versions of all bugs that the system can support, they could constitute a useful check on all new code which is input to VIPER to be used for tutoring. As well as identifying the specific version of a bug that was present, the bug-recognisers would also check that the input code conformed to the limitations on input code that VIPER's meta-interpreter requires.

9.3.3 Successful and unsuccessful aspects of VIPER's dialogues.

The tutoring goal of the system was that students should learn to associate specific viewpoints with the bugs that each one can localise. This section briefly summarises the aspects of VIPER's dialogues which the evaluation shows to be a) successful in furthering

this goal, and b) unsuccessful in this context. The 'weak' nature of the evaluation is recognised, and these findings are presented as at best 'indicative'.

The successful aspects of the dialogues were:

- The three procedural models were all relevant to describing execution, and were rigorous, clear, and suited to the needs of novices.
- These procedural models mapped clearly onto the bugs and were thus meaningful, providing a useful learning environment for novices.
- The 'Questions' menu gave the desired information in an easily understood form.
- The 'Questions' menu allowed students to investigate a bug by their own methods.
- The explanation provided by the system was generally effective in all dialogues.
- The interface was, with some reservations, adequate to the demands placed upon it.
- The dialogue structures were able to promote some learning without augmentation for diagnosis of the students.
- Learning the viewpoints was useful even for experienced users.

The unsuccessful aspects of the dialogues were:

- Some aspects of the interface, often due to the limited screen size or technical difficulties.
- Changes from applying one viewpoint to applying another were not highlighted.
- Answers given to describe the effects of a bug could not be revised.
- The lack of the full functionality of Prolog caused problems for some of the more experienced users who had to 'suspend' their knowledge of some aspects of the language.

9.3.4 System architecture.

This topic is discussed more extensively in section 9.5. Essentially, VIPER's architecture is that of a meta-interpreter, (or 'simulation') which provides a rich execution history of a

process. A range of inference procedures are then built to exploit the information in this history in relation to the different viewpoints that are to be tutored. This architecture satisfies the design goals of section 9.2 as is discussed in that section. The strategy of building a number of inference procedures on top of a rich underlying representation is similar to that used in STEAMER (Hollan et al. 1984) which is based on a rich numerical model of a steam propulsion plant, but which supports many different views of the domain. As is demonstrated below, the execution history produced by VIPER's meta-interpreter can be used to support quite different viewpoints on Prolog execution, providing that the relevant domain formulations and dialogue mechanisms are developed.

9.4 Cognitive Apprenticeship.

Chapters 2 and 3 discussed the relevance of Cognitive Apprenticeship as an educational philosophy which could be used to support the design of tutoring systems. This section briefly explores this issue in relation to VIPER.

A central tenet of Cognitive Apprenticeship is that the use and practice of knowledge to be learned should always have a central place in the tutoring system design process; see (section 2.6.2). We have tried to live up to this dictum by ensuring that each part of VIPER and its domain serve the goals of firstly describing execution clearly and unambiguously, and secondly localising bugs in a simplified domain. The results from the evaluation of chapter 8 seem to validate this approach.

The tutoring strategies of Cognitive Apprenticeship were related to the viewpoint formalisation in chapter 3 in terms of Modeling, Scaffolding, giving different problem decompositions, and general practice. VIPER's dialogues can be described as examples of these strategies. For Modeling, VIPER provides a demonstration of execution description in the first dialogue. In the bugfinding dialogues, the explanations of bugs that are offered are intended to be a demonstration of the inference procedures that can be used to identify a bug. For Scaffolding VIPER offers the three procedural models of execution and the

operators that act upon them. The possibility of different problem decompositions is made very clear by the different explanations which could be generated for the same symptom, ie. different models would imply different bugs, and even with the same model, a number of different bugs could cause the bugged behaviour. It is only the condition of having a single difference from the ideal code that allows any choice between the alternatives. As the dialogues stand, the students choose which viewpoint the bug is to be related to, but, as described above, a more general tutorial involving all three viewpoints could easily be constructed for more general practice.

In section 2.6 three kinds of 'transparency' advocated by Brown (1989) were discussed. These were Domain, Internal, and Embedding transparency. The first two of these seem relevant to VIPER. Domain Transparency is concerned with selecting a viewpoint which matches the student's goals. There can be only a limited demonstration of this in VIPER, as all the viewpoints are directed to the goal of debugging. However, the evaluation was able to demonstrate that students could select a viewpoint which matched their interest, as when the student who wished to learn about resolution, and the student who wished to learn about search chose the relevant viewpoint to work on. The system could be made much more adaptive if a wider range of viewpoints was implemented in it.

Brown's (1989) Internal Transparency describes the degree to which the models and inference mechanisms of the system match those of experts in the world, and thus the degree to which implicit learning is promoted. The intention is that students should enter the culture of the *domain* and not that of 'schoolwork'. No strong claims can be made here. Certainly, the three models of execution were designed to be used in the 'real world', and may well (when complete) be used by some practitioners for debugging, but the *social* dimension of VIPER's knowledge is non-existent. Some implicit learning may occur, as when students describing execution in the first dialogue learn when to change from one model to another. The explanations of the bugs in the bugfinding dialogues may also

promote some implicit learning of the inference procedures by which bugs may be localised.

These remarks indicate to us that Cognitive Apprenticeship is a suitable educational philosophy to support the design and implementation of tutoring systems that utilise multiple viewpoints, even though many of its demands are difficult to satisfy, and only a few have been satisfied in VIPER.

9.5 VIPER and other domains.

9.5.1 Introduction.

VIPER as described in chapter 7 does not constitute an ITS. Rather it is seen as an 'enabling technology' which could support the rapid implementation of an ITS. The purpose of this section is to show how VIPER is relevant to future work in tutoring systems. This is done by taking the solutions to the problem formulated in chapter 2, (ie. how are viewpoints to be conceptualised and how implemented?), and showing how these solutions can be generalised to another view of Prolog, and to another domain altogether. Finally, some remarks are made about a domain where VIPER's solutions could not be applied. The issue of diagnosis is not discussed here, but is dealt with in the section on future work.

Design decisions were taken at three levels during the implementation of VIPER:

- The formulation of procedural models of a subset of Prolog behaviour to enable novices to describe execution to the system.
- The formulation of a simplified domain of debugging, and the specification of 'inference operators' which mapped the procedural models onto a catalogue of bugs.
- The construction of an underlying representation for VIPER, of a meta-interpreter which produces an execution history rich enough to support:

    1) the models for execution description;

2) the models and operators for bug localisation;

3) other viewpoints on Prolog.

It is the third level of design, that dealing with the underlying representation, that is the focus of this section. The solutions exemplified here can be generalised in two ways. Firstly, the actual meta-interpreter and execution history implemented in VIPER can be used to support the tutoring of a new viewpoint on Prolog execution. The second form of generalisation is to apply the 'simulation and process-history' strategy used in VIPER to a completely new domain. Both forms of generalisation are now demonstrated. In each case the discussion deals first with the viewpoint formulations that the domain requires, and then with the underlying simulation that produces the process history to support them.

9.5.2 Another viewpoint on Prolog.

The view chosen for this exercise is that of the 'Slices' detailed by Weiser and Lyle (1986). In general terms, a slice is a partial account of an execution in terms of its data flow, which could, for instance, detail all the places where a specific variable is affected or changed. It is a more advanced view of execution than those discussed previously, and is of considerable use in debugging.

A new set of model and operators would be required to formalise the 'Slicing' viewpoint, but once this had been developed, VIPER's meta-interpreter and execution history would immediately support it without further change.

The viewpoints which utilise the process history.

The skill to be tutored in this case would be the application of an appropriate model of Prolog execution to determine at which points in the execution a specific variable is bound, unbound, or fails to be bound. The viewpoint that this application of a model formalises could be constructed in terms of the structures given in chapters two and three, ie. in terms

of a model with three classes of operators which draw inferences from it, and with some heuristics which state the viewpoint's area of applicability.

It seems likely that the 'Slicing' model would need to describe a less detailed reading of execution than the procedural models of section 6.5. These models could be used to describe 'slices' quite easily, as this would not require the special conditions given in chapter 6 to map them onto a bug catalogue in a limited domain. The important point is that the models, in combination, produce an exhaustive description of execution, and would need to be adjusted to summarise or exclude irrelevant parts of the execution. We now give an approximation of a slicing model which, in accordance with VIPER's original models, does not cater for backtracking:

1. The 'Target' variable is defined as the variable whose execution history we wish to examine, or any variable to which it is bound.
2. If the resolution of a goal literal and clausehead/fact is about to be attempted, check for the presence of the target variable in either the goal or the clausehead/fact.
3. If the target variable is present in the resolving goal and clausehead/fact,
    a) report the identity of the parent goal and resolving clausehead/fact
    b) report the details of the current head resolution;
    c) report the presence and success or failure of any subgoals.
4. If the target variable is not present in the resolving goal or clausehead/fact ignore this head resolution.

This model should produce an edited version of the code's execution history, which would only relate to resolutions of a goal literal and clausehead or fact which contained the 'target' variable, and those subgoals which confirmed or denied any bindings made in such head resolutions. The form of the operators used on such a model, and the inferences made with them, would depend on precisely how the model was to be used. If the resultant system is supposed to tutor the use of this model for debugging, then the debugging domain and the relationship of this model to it would have to be carefully designed and made explicit.

The production of the process history.

No changes would be required to VIPER's meta-interpreter, or to the execution histories that it produces, in order to support the slicing model. These already encode the unique variable identifiers used in an execution, so that an abstraction of the whole history, relating only to the resolutions, or parts of resolutions, where the specified variable was affected, would be relatively easy to produce. This is in fact the kind of activity, ('Retrospective Zooming') for which Eisenstadt (1985) developed the meta-interpretation technique used in VIPER.

9.5.3 A different domain.

The alternative domain chosen is that of WHY (Stevens, Collins and Goldin 1979), which was described in chapter 2. These authors built a tutoring system with a single, 'scriptal' view of its domain, the causes of heavy coastal rainfall. They described how a second 'functional' view of the domain would be required if the system was to cope with a number of the misconceptions that they detected in their students. One of these, the 'sponge' misconception, described rainfall as being a result of moist air being squeezed against mountains. Stevens et al. did not implement a system with this second viewpoint.

We now demonstrate how such a second view of the rainfall domain could be implemented, (along with the initial 'scriptal' view), using the design strategies developed in VIPER. This would require that both scriptal and functional models of coastal rainfall be developed, along with the operators that draw inferences on them, and, at some point, a statement of their area of application. Once this had been done, it would be possible to specify the information that had to be available in the process histories to which the viewpoints would be applied. A suitable simulation could then be built which represents the crucial factors and their interactions, and which could generate a process history in the terms required. We then show how such a system could support the distinction of the

correct and bugged accounts of rainfall, ie. the condensation as opposed to the 'sponge' account.

The viewpoints which utilise the process history.

The model for the first kind of viewpoint, that describing the "scriptal" account of rainfall, has been defined for us by Stevens et al. (1979). "Scripts" are "...generic knowledge structures..." which represent knowledge about classes of phenomena, and "...a partially ordered sequence of events linked by temporal or causal connectors". The knowledge has a hierarchical structure where the detail of one script is expanded by a subscript. The scripts describe the relationships of sets of "roles" such as "AIR-MASS" or "BODY-OF-WATER" which are bound to specific entities in the real world to describe a specific real-world process such as the evaporation of water over the Gulf Stream and it deposition as heavy rainfall on Ireland.

Examples of these scripts show a series of processes such as evaporation or movement due to wind pressure, linked by 'precedes' or 'causes' relations as in the following:

Heavy Rainfall.

1. A warm air mass over a warm body of water absorbs a lot of moisture from the body of water.

Precedes:

2. Winds carry the warm moist air mass from over the body of water to over the land mass.

Precedes:

3. The moist air mass from over the body of water cools over the land area.

Causes:

4. The moisture in the air mass from over the body of water precipitates over the land area.

(Stevens et al. 1979 p. 14 of 1982 reprint).

These examples could be readily expressed in either procedural, ('if there is a warm air mass over a warm body of water then the humidity increases over time.') or declarative ('A warm air mass over warm water gives an increase in humidity over time') form. The

knowledge represented in each script could be represented by a series of such statements which together constitute a model, and inferences made on this model as described in chapter 5. The hierarchical nature of the scripts could be represented by implementing each level as a different model, which, when combined with its own operator and heuristic set would constitute a viewpoint. The description of the evaporation process which Stevens et. al. give as a sub-script of the "Heavy Rainfall" representation, would thus be represented as a separate viewpoint, as Resolution is distinguished from Search Strategy in VIPER. A student's knowledge of such models could then be tested against the execution history stored for a given run of the simulation.

Other forms of viewpoint are necessary however, as Stevens et al.'s (1979) paper argues forcefully. Our purpose here is to demonstrate that these too could be represented by the viewpoint formalism of chapters 3 and 5. The main extra viewpoint discussed by Stevens et al. is the "functional" viewpoint, which is required in order to combat a range of misconceptions such as the 'sponge' model of precipitation and the 'cooling by contact with mountains' bug. The functional viewpoint describes a number of factors which are not intended to be causally linked or temporally ordered, but which vary in relation to each other and which produce a specific result. Condensation is described in these terms. The description (see below) is in terms of "ACTORS" which have a "ROLE" in the process and are affected by a set of "FACTORS". The "RESULT" is due to the "FUNCTIONAL RELATIONSHIP" that holds between the "FACTORS" and the "ACTORS". These relationships are expressed in a form which resembles predicate calculus. There is, for example, a positive "FUNCTIONAL RELATIONSHIP" between the "FACTOR" of the temperature of the water source, and the "RESULT" of increased humidity in the air mass. As it is stated by Stevens et al. (1979) this functional model does not describe the degree or rate of change in the factors and the result, although it may specify necessary conditions such as "WARM" for the water source. Stevens et al. (1979) give the following example of a functional model for evaporation:

Actors.

Source: Large-body-of-water.

Destination: Air-mass.

Factors.

Temperature(Source).

Temperature(Destination).

Proximity(Source, Destination).

Functional-relationship.

Positive(Temperature(Source)).

Positive(Temperature(Destination)).

Positive(Proximity(Source, Destination)).

Result.

Increase(Humidity(Destination)).

(Stevens et al. 1979 p. 16 of 1982 reprint).

The collection of statements which represent the functional knowledge could be seen as a model which acts as the core of a viewpoint as described in chapter 3. This viewpoint is completed by providing the three classes of operators to act on the model which are described in chapter 5, and a set of heuristics which govern its application.

The production of the process history.

The ordered and causally-connected process described by the scriptal model could be simulated in much the same way as the power station of chapter 4 was simulated, if the crucial variables such as temperature and humidity are defined for each of the "roles" mentioned in the scripts. In terms of Stevens et al.'s (1979) example, such a simulation would show that the longer a warm air mass was above a warm body of water, then the higher would be its humidity, and the longer it was subject to a wind from a specific direction, the closer it would be to a given land mass. If the simulation also caused the values associated with each "role" to be plotted against time, then this would provide the

sort of 'execution history' used in VIPER which could support a variety of viewpoints if it were rich enough in information. Such an execution history could thus take the form of a series of values for the relevant factors calculated and recorded by the simulation against elapsed time, ie. each value would be calculated and recorded at the end of each elapsed time interval. This could give an execution history of which the examples in table 10 could be segments. An alternative way of structuring the process history would be to record the simulation data in segments which were related to*events* in the simulated system, rather than to the passing of (simulated) time.

Table 10. Examples of possible execution history segments for a proposed 'Heavy Rainfall' tutor.

Elapsed Time: 1			
Role.	Instance.	Factor.	Value.
Body of Water	Gulf Stream	Temperature	10
Wind	South-westerlies	Strength	20 knot
Land Mass	Ireland	Precipitation	0
Air Mass		Temperature	12
Air Mass		Moisture content	30
Air Mass		Distance to Land	200 km.
Air Mass		Volume	100%
Air Mass		Altitude	0 m.

Table 10 Continued.

Elapsed Time: 50			
Role.	Instance.	Factor.	Value.
Body of Water	Gulf Stream	Temperature	10
Wind	South-westerlies	Strength	25 knots
Land Mass	Ireland	Precipitation	10
Air Mass		Temperature	2
Air Mass		Moisture content	15
Air Mass		Distance to Land	0 km.
Air Mass		Volume	200%
Air Mass		Altitude	400 m.

These execution history segments are intended to represent two widely separated stages of a specific run of a heavy-rainfall simulation. At 'Elapsed time: 1' the air mass is over a warm water mass at sea level, and being driven towards a land mass by a twenty-knot wind. The precipitation over the land attributable to the current air mass is zero. The data recorded after the next time interval, (not shown here), should show an increase in its moisture-content, with relatively constant temperature, altitude, and volume. The distance to the land mass should have decreased.

The data shown for 'elapsed time: 50' indicates that the air mass is now over the land, and that its altitude and volume have increased,while its temperature and moisture content have fallen, and precipitation has occurred.

We assume that the starting values and data of the simulation could be set to represent conditions where rainfall would occur in varying degrees, or not at all.

The scriptal viewpoint could interpret this execution history through the application of operators such as those described in chapters 3 and 5. An example of an 'inference operator' would be one which interrogated the execution history to see if the following was true:

1. The arrival of a moisture-laden air mass at the land causes it to cool.
2. Cooling of a moisture-laden air mass causes precipitation.

Therefore the arrival of a moisture-laden air mass at the land causes precipitation.

What we also require is that the execution history produced by the simulation of evaporation and rainfall should be interpretable in terms of the functional model described above. This would be possible if the "ACTORS" and "FACTORS" could be mapped onto elements in that history. This mapping need not be direct, but could be the result of whatever inferencing or transformation is required. In fact, the "ACTORS" such as "Large-body-of-water" and "Air-mass" do map easily onto the "roles" of the scriptal knowledge, as do the "FACTORS" such as their temperature and proximity. A functional interpretation of the execution history then, would require that the values of the various "FACTORS" expressed in it should vary in a relationship which does not contradict the various "FUNCTIONAL RELATIONSHIPS" described by the functional model. If, for instance, the execution history of the simulation showed a rise in the temperature of the water source, then the functional model would predict a rise in the humidity of the air mass, but would not attempt to describe any causal connection between the two events. An 'inference operator' which could made this inference would test the execution history to see if the following was true:

1. Source temperature X at time 1 and source temperature X+ at time 2 indicates a rising source temperature.
2. A rising source temperature from time 1 to time 2 indicates an increase in destination's moisture content from time 1 to time 2.

Therefore source temperature X at time 1 and source temperature X+ at time 2 indicates an increase in destination's moisture content from time 1 to time 2.

As Stevens et al. describe, the usefulness of the functional viewpoint is shown by its use in combating misconceptions. A functional model of precipitation of the type described in the scripts would describe a relationship between the altitude of an air mass, its volume, its temperature, and its ability to support water vapour. As the air mass rises, it is forced to expand and thus its temperature drops causing the precipitation. If this functional model were applied to interpret a simulation's execution history for some precipitation of this kind, then, unless the functional model is to be violated, a fall in temperature must be accompanied by a rise in altitude and an increase in volume. An attempt to explain the precipitation in terms of a 'cooling by contact with the mountains' misconception would not give the expanded volume or the increased altitude. The 'sponge' misconception, which sees the air mass squeezed against the mountains and forced to drop its moisture would give a *reduced* volume and a stable altitude. A wide range of tutorial exercises could be based on these relationships.

Two of the factors discussed in the functional model, altitude and volume, have no place in the scriptal model. It is thus clear that they would have to be specially included in the simulation of the rainfall process if the functional viewpoint on precipitation is to be supported. The cost of this seems very small in terms of the possible benefits that could be gained. The point to be made is that an ITS can be built with multiple viewpoints available to it if these are defined beforehand, and a sufficiently rich underlying representation built to support them.

The foregoing paragraphs have shown how the architecture and viewpoint formalisation used in VIPER could be applied to implement a system in another domain. The other domain chosen was the 'heavy coastal rainfall' domain of WHY (Stevens et al. 1979), one of the systems which first drew attention to the need for multiple viewpoints to be used in tutoring systems.

9.5.4 A domain that cannot be implemented using VIPER's strategies.

Since the design and implementation strategy used in VIPER relies on interpreting the execution history of some process, then any purely declarative domain such as the geography domain of SCHOLAR. (Carbonell 1970) cannot be implemented in this way. There is no process to be described and thus there is no execution history to be interpreted in terms of a model and operators. However, if we bear in mind the dictum from Cognitive Apprenticeship that the *use* of knowledge should always have a central place in the design of tutoring systems, then some possibilities for VIPER-like implementations do become apparent. The application of declarative knowledge to achieve some purpose must involve the execution of some procedure. The simulation of this procedure could give rise to an execution history which could be interpreted in the manner described above.

9.6 Future Work.

9.6.1 Introduction.

This section summarises the possible extensions and improvements that could be made to the domain formulations, the models of execution, and to VIPER itself, and where necessary indicates how these might be achieved. The section considers in turn the domain and models, the VIPER architecture, and each of the three dialogues described in chapter 7. For the architecture and for each dialogue, a sub-section considers how the system could be extended to allow the diagnosis of student errors, misconceptions, and needs. This is to be achieved using methods which have already been reported in the literature by other researchers. It is assumed that the ability to perform such diagnosis would make VIPER more adaptable to the individual student.

9.6.2 Domain Models and Domain.

The domain formulation given in chapter 6 could be expanded significantly at two distinct levels. The first of these would use the current system and execution models to implement a debugging domain for the other symptoms described by Brna et al. (1987) that were introduced in section 2.4. The two other variable-related symptoms were the unexpected failure to instantiate a variable, and the unexpected instantiation of a variable. The bug 'tree' for 'the unexpected instantiation of a variable' is given in figure 5, and a similar 'tree' could be developed for the 'failure to instantiate a variable' symptom. For VIPER to be able to tutor in relation to these symptoms, code would have to be added that could analyse execution traces with a different combination of expected and actual results than those dealt with by the present system, which expects both queries to ultimately succeed and give a variable binding. The provision for the addition of this code is described in section 7.3.2.

Other additions would also be necessary. The precise effect of each bug in relation to each symptom would have to be determined in abstract terms and stored in template form in the manner described for the 'instantiation to an unexpected value' symptom in section 6.4. This would allow critiquing of the answers given in relation to each symptom in the third dialogue described in chapter 7, where the student is asked to describe the effect and nature of each bug.

It is possible that VIPER could be extended to deal with the termination symptoms described by Brna et al. (1987), but this would certainly require a more radical re-formulation of the domain.

The second level of expansion for the domain formulation would enable VIPER to tutor in relation to a wider range of viewpoints than those detailed in chapter 6. An indication of how this could be done is given in relation to the 'slicing' viewpoint in section 9.5.2.

Other possible candidates[1] for implementation as viewpoints are:

1. Dices, (Weiser and Lyle 1986); (smaller versions of 'slices').
2. Prolog as an operator-dependent rewrite language.
3. The 'pointer' model from 'C' implementations of Prolog, where the pointers are changed to unify variables.
4. Prolog as a loop-based procedural language, using such constructs as 'do.....until'.
5. Prolog as a declarative language with search.
6. Prolog as a resolution theorem prover.
7. Prolog as having 'plastic' syntax, where a program can be data and vice versa.
8. Prolog as a rule-based system akin to productions.
9. Prolog as an object-oriented system where the clauses are objects sending messages (Kahn 1989).

In order to provide tutoring based on one of the radical reformulations given above, it would be necessary to extend VIPER, as the new viewpoint might well require information that is not at present provided by the execution history generated by VIPER's meta-interpreter. The meta-interpreter and tracing mechanisms would thus have to be augmented to provide suitable execution histories, and appropriate dialogue mechanisms would have to be provided to exploit them. The point to be made here is that radical reformulations can be tutored if they are programmed into the various system components.

The use of such alternative viewpoints could help to remedy one of the theoretical weaknesses of Cognitive Apprenticeship, that it does not allow for the occasional radical

---

[1]courtesy of Mike Brayshaw 1989, personal communication.

reformulations of the domain that actual practice sometimes demands. Others may wish to argue that Cognitive Apprenticeship does promote such reformulations in its encouragement of the search for different problem decompositions and reflection. This could only be effective if the alternative viewpoints had been previously learned.

A less radical development of VIPER's viewpoints would involve expanding the models of section 6.5 to describe the full functionality of Prolog. The added functionality would include Backtracking, embedded variables, and the use of 'cut', 'fail', 'not', and the full range of operators. Some of the changes that would be required to cater for such developments are summarised in section 6.2.6. Where backtracking was to be described, additional information would have to be included concerning the order in which subgoals are 're-done', and the order in which variable bindings are un-bound. In order to produce an appropriate execution history, the meta-interpreter of chapter 7 would have to be extended to reflect the full range of Prolog behaviours.

Two minor developments of the models and domain as they stand could lead to the resolution of some of the difficulties revealed by the evaluation described in chapter 8. One such development would be to simply remove those parts of the models which state that 'all arguments have been unified' and 'all subgoals have been proved'. They were originally included in the models to help improve their clarity by indicating when specific stages of the resolution were complete. However, several students seemed to find them tiresome and unhelpful. Were they to be removed from the models, then the code which steps through the execution history to describe an execution and to check students' descriptions would have to be adjusted to ignore the symbols relating to these model parts. The predicates which identify particular bugs and the meta-interpreter which produces the execution history would not need to be altered.

A slightly more serious difficulty is that related to the 'wrong clause order' bug. This problem is exemplified in detail in relation to question 3.4 of section 8.3.3, and is

essentially about which clause to label 'bugged' when two clauses need to have their positions swapped. At present VIPER assumes that the first clause that shows a difference to the ideal code must be the bugged one. At the point where a student has to make such decisions, (the second dialogue described in chapter 7), VIPER already 'knows' what the bug is. It would thus be a simple matter to include a predicate which only related to this bug, and which allowed either answer to the question 'which is the bugged clause' to be correct.

9.6.3 System Architecture.

Two possible developments at the system architecture level have been described elsewhere. The possibility (and the inherent difficulties) of allowing the student to make limited changes to the bugged code in order to determine the new outcome were discussed in section 9.3.1. Section 9.3.2 considered the development of the explanation predicates used in the 'bugfinding' exercises to replace the 'bug-recognisers' described in chapter 7 in order to give a more consistently 'glass-box' representation of the domain.

Diagnosis.

Diagnosis pertinent to each of the three dialogues of chapter 7 is described below. A more general level of diagnosis could be made before any of these dialogues had been initiated. For the evaluation described in chapter 8, VIPER was configured to ask students which viewpoint they wished to work on, and then choose a debugging problem related to that viewpoint. A more technically sophisticated solution would involve the system gaining information about the student's goals. If sufficient information about the use of viewpoints was available to the system, then a viewpoint suited to the student's goals could be chosen. Information about goals could be obtained most simply by asking the student about them at the start of their tutorial. Goals expressed during the evaluation involved learning more about such factors as Resolution or Search Strategy.

Other possible goals could involve learning about bugs connected with clause order, or learning to localise bugs when the bug may be related to any of the three viewpoints. If VIPER was able to determine that clause order bugs were to be localised in relation to the Search Strategy viewpoint, then this viewpoint could be revised, and a range of suitable problems chosen for the student's tutorial. All that would have to be added to the current system is the information detailing what each bug is good for, (plus the means of retrieving this) and a means of selecting and scheduling the problems. The information detailing the application of each viewpoint, (ie. what goals each one could serve), is described in chapter 3 as a necessary part of an implemented viewpoint, although this information is only implicitly present in VIPER; (see sections 9.1 and 9.2).

To serve the goal of learning to localise bugs that could be related to any of the three viewpoints, even less has to be added to the system. VIPER simply has to choose a problem and initiate the bugfinding dialogues without indicating which viewpoint the problem relates to.

Were a wider range of viewpoints to be implemented, as section 9.6.2 suggests is possible, then a much wider range of goals could be served.

As well as adapting the tutoring to a student's goals, it could be adapted to their experience, assuming that information about this could be obtained. The evaluation showed examples of users who had written theorem-proving programs in other languages, and who deliberately chose to work on the Resolution viewpoint as this was most familiar to them. Where the system is dealing with a complete novice, then such information could be very useful in determining which model of execution should be tutored first to introduce Prolog.

9.6.4 Dialogue 1: Execution Description.

A number of possible developments have been suggested for this dialogue at different points in the thesis. They are summarised here, along with some possibilities that have not been mentioned previously.

The possible extensions to dialogue 1 are as follows:

- The facility which automatically provides an explanation of the correct execution step in response to mistaken student input can be switched on or off. Control of this switching could be given to the student via a menu option.
- At present when a new resolution is to be attempted, the student merely indicates this fact via menu choices. The execution history actually stores the number of the clause with which the goal is to be resolved, and the student could be asked to supply this as a part of the execution description.
- It was evident from the evaluation that some students forgot which clause was being resolved with the current goal, even though the number of this clause was displayed in the dialogue window. As this number is available to the dialogue predicates, the relevant clause in the code listing could be highlighted as an aid to memory while the students concentrate on the details of the execution. Predicates for performing this operation are provided by the LPA MacProlog environment.
- In order to popup the relevant execution history menu to make a selection from it, students have to click on a button to indicate which model they wish to apply for each step of the execution. Some novices found the choice of models for application confusing, and asked for assistance to be provided with this. Such assistance could be provided through the predicates which grey-out or 'disable' buttons if both of the non-applicable buttons were greyed out.
- The mechanism described in the previous point could also be used to indicate which model was currently being applied if the non-applicable buttons remained 'disabled' while the dialogue relating to a particular step was being conducted.

• When execution description involves a change from applying one model to applying another, some explanation of this change could be given indicating why the change was necessary. Such explanation would help students to understand *when* the different models should be used for execution description. The explanation could be provided through text templates similar to those which currently explain the individual execution steps. The choice of template for presentation to the student could be keyed to the pattern of execution symbols in the execution history which corresponded to the change of viewpoint.

Diagnosis

The version of Dialogue 1 described in chapter 7 involves the use of all three models of section 6.5 to describe the total execution. Since the responses to student input are dependent on an analysis of the symbols contained in each segment of the execution history, it would not be difficult to limit the dialogue to describing the execution in terms of a single model eg. Resolution. Steps in the execution history with symbols relating to other models than the selected one could simply be ignored.

Such a strategy could be adopted in response to:

- a student's requests or goals;
- a student's errors;
- a student's learning history.

Adapting the tutoring in this way in response to the student's goals would require some form of modeling such as that described in section 9.6.3. Adapting it in response to the student's errors would require at least the minimalist numerical modeling described in section 7.4.2, which keeps a numerical record for each model part of potentially correct, correct, and wrong answers. If a particular model or model part showed a particularly high score for wrong or missed-potentially-correct answers, then it would seem to be a sound

tutorial tactic to focus attention on that model by excluding the other two from the execution description.

VIPER at present keeps no record of a student's learning history. However, where a novice was being introduced to the domain and the three models of execution, such a model could be useful, and could be implemented in terms of the 'overlay' modeling first used in SCHOLAR (Carbonell 1970). Such a model would record those aspects of the models with which the student was known to be familiar, and the description of execution could then be limited to those aspects. Such a model, possibly coupled with the minimal numerical model described above, could also be used to determine when a student had a sufficient knowledge of the models used for execution description to progress to the dialogues dealing with the localisation of bugs.

A more sophisticated form of diagnosis could relate to the nature rather than the quantity of erroneous execution steps chosen by a student. A comparison of the correct step with the one actually chosen may well yield information about misconceptions in the student's knowledge of the domain. The efficacy of this technique has been demonstrated with such systems as BUGGY (Brown and Burton 1978) and DEBUGGY (Burton 1982), although it does not attempt to deal with the issue of *why* a student has a particular misconception. We are not proposing an exercise of the complexity of DEBUGGY, which attempted to diagnose multiple interacting bugs, and also the inconsistent application of a bugged procedure. What is proposed here is an enumeration of the well-known bugs in describing Prolog execution, and an attempt to match these to the student's input.

BUGGY attempts to achieve such a match by assembling fine-grained sub-skills until a model which matches the student's performance is achieved. An extended VIPER could attempt the same matching by an analysis of the sequences of execution symbols that a student's input represented. A range of common bugs are defined for us by Fung et al. (1987) and Taylor (1988), although the exploitation of these would require a meta-

interpreter and tutorial capability which reflected the full functionality of Prolog. An example of a misconception which could be detected in this way is the 'facts first' misconception detailed by Fung et al. Students with this misconception assume that Prolog immediately attempts to resolve a goal literal with a fact that could prove it, rather than with any other clauses of compatible functor and arity that are prior to the fact in the database. If, as suggested above, students were asked to indicate which clause was next to be resolved with the current goal literal, then the repeated skipping of compatible clauses could indicate that this misconception was present.

Many of the misconceptions catalogued by Fung et al. relate to backtracking and the use of the cut. If VIPER's models and mechanisms were upgraded to describe these aspects of Prolog, then it is likely that many more misconceptions could be 'caught' in the manner described above. In fact, the current execution pattern described by VIPER would then become a perfect specification of the 'try once and pass' misconception which assumes that a resolution fails completely whenever a subgoal first fails.

9.6.5 Dialogue 2: Identifying the bugged clause.

The evaluation of chapter 8 gave rise to suggestions for two minor augmentations which could improve the clarity of this dialogue for students, by providing more support structures for them. The first of these was that the dialogue should provide some statement of the viewpoint that was currently chosen for the exercise, and of the possible bugs associated with it. This would constitute the explicit statement of the area of application of each viewpoint that was discussed in section 9.2. If more screen space were available, then the problems involved in providing such information would be trivial.

The second suggested augmentation would be equally simple to implement, and would also require additional screen space. This calls for a re-statement of the conditions under which the bugs and their mapping onto the models are defined.

A more ambitious project would be to tutor directly the inference operators associated with each viewpoint, rather than just using them to structure the explanations given in Dialogues 2 and 3. Such a tutorial tactic follows rationally from the insistence that these operators are an essential element of a viewpoint, but it is not clear to what extent they actually reflect the methods used by students or practitioners in the domain, and thus to what extent they would be useful to students. Empirical work could be undertaken to investigate this topic.

Diagnosis.

A diagnostic possibility for this dialogue is to monitor the student's use of the 'Questions' menu to determine whether the questions being asked are relevant to the current bug, and whether they reflect an accurate application of the operators embodied in the explanations. Considerable difficulties are foreseen here. One of the comments made about the 'Questions' menu during the evaluation was that it allowed students to explore the bug in their own way. If this is in fact a positive quality of the dialogue, then it would seem perverse to impose the analytical methods favoured by the system as being the 'correct' path to a solution. There is also a more practical problem. If a set of bugged code contained five clauses, then a student may choose to ask questions about the four that did *not* contain the bug in order to eliminate them, before asking about the one clause that is suspected. How is the system to distinguish this purposeful activity from lost meandering?

9.6.6 Dialogue 3: Describing the bug's effects.

As with the other dialogues, a number of minor augmentations are considered here before dealing with more ambitious ideas.

The suggested minor developments are as follows:

- Make the different menus for each template slot, ('functor', 'arity' etc.) pop up automatically when the previous slot had been filled. This mechanism could

possibly be governed by some unsophisticated modeling which would ensure that the student has already learned the sequence of descriptions to be made in this dialogue.

• Allow the revision of input given as answers to previous slots in the dialogue. As pointed out in section 8.4.2, this would necessitate that the consistency of the entries in slots subsequent to the revised one would have to be checked, or new input requested.

• Add a predicate to this dialogue which determines which part of a bugged execution is to be described. In the case of the 'variable instantiated to the wrong value' symptom the questions could be:

a) where and how does the 'ideal' binding fail to occur?

b) where and how is the 'bugged' binding made?

• Add a question to the descriptions of the head matching asking for details of any variable bindings that take place there. The information required to check the accuracy of answers to this question is already present in the execution history created by VIPER. All that would be required to implement the development is a predicate which would include the relevant data in the template which describes the effect of the bug.

Diagnosis.

The most necessary development in relation to this dialogue is that which would allow some critiquing of the answers that the student gives as input to the different slots of the dialogue. The student's answers can be analysed in terms of their internal consistency, and also in terms of their relationship to the actual execution histories. After some discussion, we conclude that any analysis of student input to the Search Strategy and Code Error slots should concentrate on the student's knowledge of the mapping from viewpoints to bugs. Where other parts of the dialogue reveal difficulties related to the description of execution, these should be remedied using a version of the execution description dialogue.

The first answers a student gives in this dialogue describe the matching of a clause head and goal literal. These have to be internally consistent, as VIPER will not otherwise accept them. Repeated unacceptable answers to this section could be taken to indicate that revision of the Resolution model was required. The answer to the Search Strategy slot requires the student to indicate whether a new clause is sought for resolution, or an attempt made to complete the proof with the current clause by dealing with any subgoals. If the head resolution has succeeded, then the possibility of proving subgoals must be considered. Answers that suggest moving on to a new clause when the head resolution has succeeded are at present rejected by the system, but could be taken as an indication that the Search Strategy model needs to be re-tutored. Answers that relate to the presence of subgoals when in fact there are none, or which do not relate to them when they *are* present, seem to indicate an error so gross as to be uninformative in this case.

Answers that give inaccurate predictions of the success or failure of subgoals that are present would indicate an inaccurate application of the three models of execution. The augmented system could deal with this by reverting to an execution-description dialogue which started with the resolution of the current goal literal and clause in the context of the other clauses used to obtain the bugged result. If the student makes mistakes in describing the execution, these should reveal the misconceptions in their models of execution, and could be diagnosed as described in section 9.6.4.

The student's answer to the Search Space slot begins the process of mapping the specific execution description onto the bug catalogue. It is their knowledge and application of this mapping that must now be assessed.

The first level of this assessment concerns the linking of specific bugs with specific viewpoints as described in section 6.4.3. A Search Space entry of 'Ok' must be followed by a Code Error relating to either the Search Strategy or the Resolution viewpoints. Where the Search Space entry is one of the Search Space bugs, then the Code Error must also be

this bug, given the 'only one difference' condition described in chapter 6. Answers that do not conform to these requirements indicate that the students have not understood the conventions that underlie VIPER's domain formulation. This could be remedied by tutoring and exercises which revise and explain these conventions.

A more profound level of assessment would relate the (hopefully consistent) student's answers to the actual executions recorded in the execution histories and to the actual difference between the ideal and bugged code. Where their description of the head resolution and Search Strategy differed from the actual pattern of execution, then this would again indicate a failure to correctly apply the execution models, and could be dealt with as described above.

The student's statement concerning the Search Space and Code Error identifies a specific member of the bug catalogue as being present in a specific clause, and the accuracy of this can be checked by reference to the description template assembled by the predicates which initiate this dialogue. Where the student identifies the wrong Code Error, the system could easily be augmented to show the student the relevant parts of the bugged and ideal clauses concerned, along with a statement to the effect that the difference described by the student did not in fact exist. This statement could also suggest that the Code Error described by the student should have led them to form a hypothesis which could have been checked using the 'Questions' menu. The system could be equipped to demonstrate this by defining predicates for each bug which state the implied condition of the ideal code, and retrieve the information required to check whether it is true. The predicates needed to extract the information which would show the students that they have made a mistake, or which would check the generated hypotheses, are already implemented to serve the options available on the 'Questions' menu.

A more subtle reaction to a student's mistake at this level would be to determine whether the change implied by the Code Error they name would in fact produce the desired result in

the bugged code. Implementing such an approach would create a number of problems. The student would have to be prompted to specify exactly what change they would make to rectify the Code Error they have described, and the system would have to be able to check that the proposed change was within the constraints of the bug catalogue and the conditions under which the different code executions are analysed. If the student's answer did not produce the desired result, then a second round of execution description would have to be undertaken for the changed code to determine why it behaved as it did. This would most likely cause the student to forget or lose sight of the original execution being described in the dialogue. This exercise also seems to defeat the point of only allowing a single difference between the ideal and bugged code, which was to define a simple and tractable domain for both system and student.

We conclude that diagnosis carried out on the student's input to the Search Strategy and Code Error slots of this dialogue should concentrate on the student's mapping of viewpoints onto bugs, as this mapping was designed to be simple, and is, in essence, the knowledge that the system was designed to tutor. There seems to be little point in supporting complex diagnoses in an artificially simplified domain. Where difficulties or misconceptions related to the description of execution are diagnosed, they should be dealt with in a version of the execution description dialogue.

9.7 Conclusions to Chapter 9.

Section 9.1 considers the goals set for the viewpoint formalism in chapters 2 and 3 in the light of the subsequent implementation. It is concluded that a number of these goals have been successfully reached, and that the formalism is thus a useful one. A system has been built which utilises multiple viewpoints to carry out a single task, and which employs two of the classes of operators described in chapter 5. These viewpoints allow the system to perform the relevant tasks in the domain to be tutored, and conform to the description of referring to the same set of objects, but identifying and structuring them in different ways.

Section 9.2 considers design goals for VIPER detailed in chapter 3. These goals relate the use of multiple viewpoints to tutoring, explanation and demonstration in the chosen domain. VIPER successfully tutors each viewpoint independently, and makes their area of application clear implicitly. VIPER does not tutor the relationship between the viewpoints. VIPER is able to provide explanation and demonstration for each of the three dialogues described in chapter 7.

Section 9.3 discusses the actual design of VIPER in terms of debugging, black-box verses glass-box solutions, explanation, the dialogue structures and the system architecture. The execution description and debugging tasks of VIPER are related to such tasks in the 'real world'. It is concluded that execution description in VIPER is directly equivalent to the 'real world' task, although it is limited to a subset of Prolog behaviour. In relation to debugging, it is maintained that the execution models used in VIPER are authentic even if the simplified domain is not. VIPER is described as *embodying* a theory of debugging rather than directly tutoring such a theory. The possible difficulties involved in attempting to implement such a theory for tutoring are explored, and the claim made that the decision to use a simplified domain was vindicated.

In section 9.3.2 the issue of black-boxes and glass-boxes is discussed in relation to VIPER's design. Both forms were used, with the glass boxes intended to provide explanation. This explanation provision was partially successful. The glass boxes could be developed to carry out all system functions and thus eliminate any redundancy.

Section 9.3.3 summarises the successful and the unsuccessful aspects of the three dialogues described in chapter 7, while section 9.3.4 draws attention to the 'meta-interpreter and execution history' architecture used in VIPER.

The relevance of Cognitive Apprenticeship as an educational philosophy is the focus of section 9.4. We claim to have followed the dictum that the use and practice of the

knowledge to be learned should have a central role at all stages of the design process, and discuss the means by which the tutorial tactics of modeling, giving scaffolding, emphasising different problem decompositions and then general practice are realised in VIPER. Our success in achieving Brown's (1989) Domain and Internal transparency is discussed, showing greater success with the former than with the latter.

Section 9.5 demonstrates how VIPER is relevant to future work in tutoring systems. The solutions used in VIPER are generalised to another view of Prolog, and to another domain altogether. A domain where VIPER's solutions could not be applied is identified. This generalisation is done in terms of VIPER's underlying representation of a 'simulation' and execution history. The implementation of Weiser and Lyle's (1986) 'slicing' viewpoint is demonstrated for our Prolog behaviour subset using the viewpoint formalism of chapters 3 and 5 and the point is made that VIPER's meta-interpreter and execution history production predicates would not have to be altered.

The section goes on to demonstrate how the viewpoint formalism and design strategy used for VIPER could be applied to implement the system described but not implemented by Stevens, Collins and Goldin (1979). This system would have been an extension of the WHY system, and would have described heavy coastal rainfall in terms of 'scriptal' and 'functional' viewpoints. This demonstration involves defining examples of scriptal and functional viewpoints in terms of the formalisation of chapters 3 and 5, and producing an outline specification for a simulation of the rainfall-producing process and the execution histories that it would create. In this case, none of VIPER's actual components could be used. Section 9.5.4 notes that systems using multiple viewpoints for purely declarative domains cannot be implemented in this way.

In section 9.6 a wide range of future work is considered. This deals with possible improvements and extensions to the domain models and the domain formulation, and to VIPER itself. The possible work on VIPER is described in terms of the system

architecture, and each of the three dialogues described in chapter 7. Each of these aspects of the system is discussed firstly in relation to general improvements or extensions that could be made, and then in terms of any diagnostic capabilities which could be added to the system. A range of diagnostic possibilities and methods for implementing them are suggested.

Chapter 10. Conclusions.

Chapter 9 discussed VIPER and its design in detail. This chapter states our general conclusions in relation to the research goals stated in chapter 3, and the pedagogical goals which may be served by the implementation of multiple viewpoints in a tutoring system.

10.1 Research Goals.

10.1.1 The viewpoint formalism.

The goals stated in section 3.2 required us to find a suitable representation for formalising viewpoints in order for them to be implemented in a tutoring system in such a way that the system could itself perform the domain tasks. Rather than identify research on viewpoints with the intractable area of research on belief systems, we decided to investigate the question of how a tutoring system might be designed to use a range of *pre-defined* viewpoints, (see section 2.1.6), and what could be achieved by such an exercise. In order to represent these kinds of viewpoints we developed the formalism of a model interrogated by operators which produced varying inferences from the same model. Three classes of such operators are defined in section 5.3. In VIPER, three different viewpoints were implemented using three different models, with operators from the first two of the operator classes. These viewpoints were found to be of great utility in enabling the system to perform tasks in the domain and to provide explanations of those tasks.

The models themselves were tutored to the students through VIPER's critique of their description of execution, and found to be a very clear and rigorous basis for understanding the subset of Prolog behaviour that they represented. The discussion of further work indicates how the viewpoint formalism could be applied both to other views of Prolog and to other domains altogether.

The utility of the model-and-operators formalism for the representation of viewpoints was also tested through its use in a protocol analysis study. Here, it was used to formalise the inferences displayed by two groups who had been asked to control a simulated system. One of the groups was asked to use a functional model, and the other a structural model. This formalisation used all three classes of operators described in section 5.3, and was able to deal with the wide variety of inferencing displayed by the system controllers.

The third class of operator, that which changes the central model by adding or deleting information, was not tested in the implementation of VIPER, so that no further conclusions concerning its utility for this purpose can be drawn here.

The final component of the viewpoint formalism proposed in chapter 3 is a set of heuristics which together describe the area to which the relevant viewpoint may be applied. The utility of this component was not tested in the protocol analysis study described in chapter 4, as the respondents involved were *instructed* about which viewpoint to use. We had at this point concluded that it was necessary to develop a detailed formulation of the model and operators before investigating descriptions of how they were to be employed.

This heuristic component was not explicitly implemented in VIPER, but was of fundamental importance in the formulation of the simplified debugging domain. This formulation mapped the three models of execution onto the restricted bug catalogue in such a way that each bug was clearly identified with a particular viewpoint. This aspect of the design was shown to be significant in the evaluation of VIPER, where several users clearly indicated that it helped them to grasp the fact that particular bugs could be localised with particular viewpoints.

We conclude from these remarks that the use of a model and the first two classes of associated operators from section 5.3 is a very successful method of implementing pre-defined viewpoints for tutoring systems. The third class of operator may promise much

but its utility in an implementation has not been demonstrated. The use of heuristics which detail a viewpoint's area of application has not been demonstrated in terms of an explicit representation, but has been shown to be of great importance for the formulation and tutoring of a domain.

10.1.2 The system architecture.

The performance goals of VIPER stated in section 3.2 were that it should be able to tutor in a specific domain using two or more viewpoints on that domain. This required that there should be a system architecture and a domain representation which could support each of the viewpoints to be tutored, assuming that the viewpoints were implemented in the manner described in section 3.3.

The solution to this requirement has been demonstrated in the domain of Prolog debugging for novices in the form of a meta-interpreter which produces a very detailed execution history for each execution. This execution history supports tutoring in relation to execution in terms of the Resolution, Search Strategy, and Search Space viewpoints. This involves the use of a variety of model and operator combinations. When execution histories for ideal and bugged code sets are compared and analysed, the architecture also supports tutoring in terms of these three viewpoints in relation to a simplified debugging domain.

The discussion of further work has indicated how this tutoring could be made much more adaptive, and also how the same meta-interpreter and execution history used in VIPER could be used to support tutoring in relation to other views of Prolog. The discussion also indicated how the same design strategy could be used to implement a version of WHY (Stevens et al. 1979) which actually did utilise the two viewpoints described by Stevens et al. This implementation would use a simulation and process history to tutor the domain of heavy coastal rainfall in terms of scriptal and functional viewpoints.

We thus wish to claim that the strategy of using a meta-interpreter or simulation to produce a history of some process which is sufficiently detailed and structured to support tutoring in terms of multiple viewpoints on the relevant domain, is a robust and flexible means of implementing systems which can tutor in terms of multiple pre-defined viewpoints. This method is most suited to domains where distinct processes can be described.

10.1.3 Cognitive Apprenticeship.

Sections 2.6 and 3.3.2 discussed the suitability of Cognitive Apprenticeship as an educational philosophy which could support the design of systems which tutored with multiple viewpoints. This philosophy implied that the systems should have an 'authentic' view of the domain, and that the end use of the knowledge being taught should be considered at all stages of design. It suggests the tutorial strategies of modeling, giving scaffolding, asking for different problem decompositions, and giving general practice, and emphasises the social dimensions of situated knowledge. The extent to which VIPER achieves these goals is discussed in section 9.4, and it is concluded that partial success has been achieved in relation to the provision of authentic domains, and that the use of the knowledge was indeed considered at all stages in design. The tutorial strategies suggested effective forms of dialogue which were, or could easily be, implemented in VIPER. The greatest omission in VIPER's design is in relation to the social dimensions of the knowledge being taught.

These points lead us to conclude that Cognitive Apprenticeship is an educational philosophy well-suited to the design of multiple-viewpoint systems, as it makes a clear contribution to all stages of the process. At the stage of domain formulation, the philosophy establishes the criteria that the formulations must be relevant to those used by practitioners in the real world, and also to the student's intended use of the knowledge. For system design and implementation the philosophy suggests a range of effective interrelated tutoring strategies, and sets the design goal of providing an environment that is

as close as possible to that found in the real world. In emphasising the social aspects of knowledge, the philosophy suggests a direction for future research and development which promises increasingly effective systems.

10.2 Pedagogical goals served by the use of multiple viewpoints.

This research is motivated by goals which are ultimately pedagogical. We wish to build systems which support learning. It is in this context that the problem of using multiple viewpoints was originally identified. This being so, we must summarise the pedagogical goals that VIPER has shown to be served by the use of multiple viewpoints.

10.2.1 Adaptation to the goals of the student.

The evaluation detailed in chapter 8 demonstrated that students found it very useful to be able to structure their interaction with VIPER by selecting a viewpoint which matched their goals (eg. learning more about Resolution) and which formed the basis of the exercises. The possibility of the system being augmented to adapt its tutoring by itself selecting the viewpoint is discussed under the title of 'Future work' in section 9.6. The point is made that this selection could also be made on the basis of the student's experience. This facility would become more significant were a wide range of viewpoints to be implemented. A list of such viewpoints is given in section 9.6.2, and an example of how they might be implemented is given in section 9.5.2. The availability of such a range of domain formulations would make it clear that neither system nor student could simply rely on one formulation of the domain. It would thus be necessary for both parties to pay attention to the utility of each viewpoint, ie. to the meta-knowledge which detailed how each formulation was to be used. This process was observed in the evaluation described in chapter 8, where users reported a strong appreciation of the kinds of bugs that could be localised with each viewpoint. We thus wish to claim that the use of multiple viewpoints

facilitates adaptation to the student's goals or experience, and can promote learning in relation to the meta-knowledge relevant to each viewpoint.

10.2.2 Explanation in terms of different viewpoints.

One of the goals stated in chapter 3 was that the viewpoint formulation should enable the system to perform the different domain tasks that it tutored. This goal was realised, so that VIPER can describe and demonstrate execution, identify the bugged clause and the bug, and describe the effects of the bug on execution. This ability to perform each of the tasks in terms of the viewpoints it is tutoring allows VIPER to provide explanation and demonstration of each task in terms of each viewpoint. The system structures that support this are described in chapter 7. The use of multiple viewpoints thus allows the provision of explanation in terms of each one.

10.2.3 Sub-dividing the domain.

The execution models divided the subset of Prolog that they described in to smaller, distinct and clearly-described sub-divisions that had several beneficial effects for the student that are apparent in the evaluation of chapter 8. The different models, and the parts that constituted them, could be learned incrementally, so that individual learning tasks were small, and, when accomplished, showed clear progress. The identification and clear exposition of a range of viewpoints also helps students to themselves determine the areas in which they are weak, and thus to structure and focus their learning activities.

10.2.4 The elimination of misconceptions.

This is not a topic about which we can make strong claims based on VIPER's implementation and evaluation. The problem was introduced in the context of using multiple viewpoints by Stevens et al. (1979), who described how the use of a 'functional' viewpoint could help to rectify misconceptions which arose when tutoring was

accomplished using a single 'scriptal' viewpoint. In section 9.5.3 we have indicated how a system using VIPER's design strategy could be built to support and tutor the two viewpoints on heavy rainfall of which were described by Stevens et al. (1979), although they only implemented one of them. We also indicated how the process history produced by such a system could be interpreted in terms of the functional viewpoint so as to combat the 'cooling by contact' and 'sponge' misconceptions. The construction of such a system is not seen as inherently problematic.

In chapter 9.6.4 we describe how VIPER could be augmented through the use of a technique which had been previously described by such authors as Brown and Burton (1978) to recognise 'bugs' in the student's knowledge and procedural skills. The previous work shows this augmentation to be quite feasible, if of limited value. This augmentation would allow some misconceptions in the student's knowledge of the domain to be recognised and described in terms of the system's viewpoints. Appropriate tutoring could also be given in terms of these viewpoints.

If the meta-knowledge relating to the application of each viewpoint was represented explicitly in VIPER, this would clarify the relationship between viewpoints and allow the diagnosis and correction of problems which arose from their mis-application.

Our claim here must thus be that, were further work to be undertaken, the implementation of multiple viewpoints could help VIPER to detect and remedy some misconceptions in the student's knowledge of the domain.

10.3 Summary of Claims.

The strong claims made in chapter 10 may be summarised as follows:

- The use of a model and the first two classes of operator described in section 5.3 is a very successful method of implementing pre-defined viewpoints for tutoring systems. The heuristics detailing a viewpoints area of application have been shown to be of great importance in the formulation and tutoring of a domain.
- The strategy of using a meta-interpreter or simulation to produce a detailed history of a process and of using this history to support tutoring in terms of multiple viewpoints on the relevant domain is an effective method of implementing systems which can tutor in terms of multiple pre-defined viewpoints. This method is most suited to domains where distinct processes can be described.
- The educational philosophy of cognitive apprenticeship is well-suited to support the design of tutoring systems that use multiple viewpoints on their domains.
- The use of multiple viewpoints facilitates the system's adaptation to the student's goals and experience.
- The use of multiple viewpoints allows explanation to be provided in terms of each one.
- The identification and careful tutoring of a range of viewpoints helps students to determine their own areas of weakness, and thus to structure and focus their learning.

References.

Anderson J.R. and Reiser B.J. (1985). "The LISP tutor". Byte vol. 10, no. 4, pp.159-175.

Brna P., Bundy A., Pain H. and Lynch L. (1987). "Programming tools for Prolog environments", in Hallam J. and Mellish C., (Eds.), Advances in Artificial Intelligence, pp. 251-264. John Wiley and Sons.

Brna P., Bundy A., Pain H., and Lynch L., (1987b). "Impurities in the Proposed Prolog Story". Working Paper no. 212 Dept. of Artificial Intelligence, Edinburgh University, Edinburgh, UK.

Brna P., Bundy A. and Pain H. (1988). "Prolog: a survey of available tools". DAI Technical Paper no. 3, Dept. of Artificial Intelligence, University of Edinburgh, Edinburgh, UK.

Brown J. S. and Burton R. R. (1978). "Diagnostic models for procedural bugs in basic mathematical skills". Cognitive Science vol. 2, pp. 155-191.

Brown J. S. and Burton R. R. (1975) "Multiple Representations of Knowledge for Tutorial Reasoning", In Bobrow D. and Collins A. (Eds.), Representation and Understanding: Studies in Cognitive Science. Academic Press, New York.

Brown J. S., Burton R. R. and de Kleer J. (1982) "Pedagogical, natural language, and Knowledge engineering techniques in SOPHIE I, II, and III." in Sleeman D. H. and Brown J. S. (Eds.) Intelligent Tutoring Systems. Academic Press, London.

Brown J.S. Burton R.R. and Zdydel F. (1973). "A model-driven question-answering system for mixed-initiative computer-assisted instruction". IEEE Transactions on Systems, Man, and Cybernetics, vol. 3, pp. 248-257.

Brown J.S., Collins A. and Duguid P. (1989). "Situated Cognition and the Culture of Learning". Educational Researcher vol. 18 no. 1 pp. 32-42.

Brown J. S. Rubinstein R. and Burton R. R. (1976) "A Reactive learning environment for computer-assisted electronics instruction". BBN Report 3314, Bolt, Beranek and Newman Inc., Cambridge, Mass.

Bundy A., Pain H., Brna P. and Lynch L. (1985). "A proposed Prolog story". D.A.I. Research Paper no. 283, University of Edinburgh, Edinburgh, UK.

Burton R.R. (1975). Semantically-centred Parsing. Doctoral dissertation, Univ. of California, Irvine, California.

Burton R.R. (1976). "Semantic grammars: an engineering technique for constructing natural language understanding systems". BBN Report 3453, (ICAI Report 3), Bolt Beranek and Newman Inc., Cambridge, Mass. U.S.A.

Burton R.R. (1982). "Diagnosing bugs in a simple procedural skill", in Sleeman D.H., and Brown J.S., (Eds.), Intelligent Tutoring Systems, Academic Press, London.

Burton R.R. and Brown J.S. (1979). "An investigation of computer coaching for informal learning activities". Int. J. of Man-Machine Studies. 11, pp. 5-24. Reprinted in

Sleeman D. and Brown J.S. (Eds.) (1982) Intelligent Tutoring Systems. Academic Press, London.

Byrd L. (1980). "Understanding the control flow of Prolog programs", in Tarnlund S. (Ed.), Proceedings of the Logic Programming Workshop, Debrecen, Hungary, 1980.

Carbonell J.R. (1970). "AI in CAI: an artificial intelligence approach to computer-assisted instruction". IEEE Transactions on Man-Machine Systems, vol. 11, no. 4, pp. 190-202.

Clancey W.J. (1979). "Tutoring Rules for guiding a case-method dialogue". Int. Jrnl. Man-Machine Studies. vol.11, pp. 25-49.

Clancey W.J. (1983). "The Epistemology of a Rule-Based expert System - a Framework for Explanation.", in Artificial Intelligence vol. 20, No. 3.

Clancey W.J. (1985). "Heuristic Classification", Artificial Intelligence 27, pp. 289-350.

Clancey W.J. (1987). Knowledge-Based Tutoring. The GUIDON Program. MIT Press.

Collins A., Brown J.S. and Newman S. (1987). "Cognitive Apprenticeship: teaching the craft of reading, writing and mathematics" in Resnick L.B. (ed.) Cognition and Instruction. Lawrence Erlbaum, Hillsdale, NJ.

Cumming G. and Self J.A. (1989b). "Intelligent educational systems: identifying and decoupling the conversational levels". In press.

de Kleer J. and Brown J.S. (1983). "Assumptions and ambiguities in mechanistic mental models" in Gentner D. and Stevens D.L. (Eds.) Mental Models. Lawrence Erlbaum Associates, Hillsdale, NJ.

di Sessa A. A. (1986). "Models of Computation", in Norman D. A. and Draper S. W. (Eds.) User Centered System Design. Lawrence Erlbaum Associates, Hillsdale, NJ. and London.

du Boulay B., O'Shea T. and Monk J. (1981). "The black box inside the glass box: presenting comnputing concepts to novices." Int. J. Man-Machine Studies. vol. 14, pp. 237-249.

Eisenstadt M. (1984). "A Powerful Prolog trace package" in Collected HCRL papers from ECAI-84: the 6th. European Conference on Artificial Intelligence, Pisa, Italy, Sept. 1984, HCRL Tecnical Report no. 10, HCRL, The Open University, Milton Keynes, UK.

Eisenstadt M. (1985). "Tracing and debugging Prolog programmes by retrospective zooming". HCRL Technical Report no. 17, The Open University, MiltonKeynes, UK.

Fagin R. and Halpern J.Y. (1987). "Belief, awareness, and limited reasoning." Artificial Intelligence. vol. 34, pp. 39-76.

Fung P., du Boulay B. and Elsom-Cook M.T. (1987). "An initial taxonomy of novices' misconceptions of the Prolog interpreter". CAL Research Group Technical Report no. 69, IET, The Open University, Milton Keynes, UK.

Gentner D. and Gentner D.R. (1983). "Flowing Waters or Teeming Crowds: Mental Models of Electricity", in Gentner D. and Stevens D. L. (Eds.) Mental Models. Lawrence Erlbaum Associates, Hillsdale, NJ.

References

Gentner D. and Stevens D.L. (Eds.) (1983). Mental Models. Lawrence Erlbaum Associates, Hillsdale, NJ.

Hollan J. D., Hutchins E. L. and Weitzman L. (1984). "STEAMER: an interactive inspectable simulation-based training system." A.I. Magazine. vol. 5, no. 2, pp. 15-27.

Johnson W.L. (1987). "Modeling programmers' intentions" in Self J.A. (Ed.) Artificial Intelligence and Human Learning: intelligent computer-aided instruction. Chapman and Hall, London.

Johnson W.L. and Soloway E.M. (1984) "PROUST: knowledge-based program debugging". Proceedings of the Seventh International Software Engineering Conference, Orlando, Florida, pp. 369-380.

Johnson-Laird P.J. (1983). Mental Models. Cambridge University Press.

Kahn K.M. (1989). Objects - A fresh look. Research paper, Xerox Palo Alto Research Centre, 3333 Coyote Hill Road, Palo Alto Ca. 93304.

Kieras D.E. (1984). "A Simulation Model for proceedural inference from a mental model for a simple device". Technical Report.No. 15, UARZ/DP/TR-84/ONR-15, University of Arizona, Dept. of Psychology, Tucson, U.S.A.

Kieras D. E. and Bovair S. (1984). "The role of mental models in learning to operate a device". Cognitive Science. 8, pp. 255 - 273.

Kornfeld W.A. and Hewitt C. (1981). "The scientific community metaphor". IEEE Transactions on Systems, Man and Cybernetics, 11, pp. 24-33.

Larkin J.H., McDermott J., Simon D.P. and Simon H.A. (1980), "Expert and novice performance in solving Physics Problems", Science vol. 208 pp.1335-1342.

Lesgold A. (1988). "Toward a theory of curriculum for use in designing intelligent instructional systems" in Mandle H. and Lesgold A. (Eds.) Learning Issues for Intelligent Tutoring Systems", Springer-Verlag, New York, Berlin.

Martins J.P. and Shapiro S.C. (1988). "A model for belief revision". Artificial Intelligence, 35, pp. 25-79.

McKeown K.R. (1988). "Generating goal-oriented explanations". International Journal of Expert Systems, vol. 1 pt. 4. pp. 377-395.

Minsky M. (1981). "A Framework for representing knowledge." in Haugeland J. (Ed.) Mind Design, The MIT press, Cambridge MA, pp. 95-128.

Moyse R. (1989). "Knowledge Negotiation implies multiple viewpoints", in Bierman D. Breuker J. and Sandberg J. (Eds.). Artificial Intelligence and Education: Proceedings o the 4th. International Conference on AI and Education, Amsterdam, Netherlands, pp. 140-149.

Moyse R. (1991). "Multiple viewpoints imply Knowledge Negotiation". Interactive Learning International, vol. 7 no. 1 in press.

Newell A. and Simon H.A. (1972). Human Problem Solving. Prentice-Hall, Englewood Cliffs, New Jersey, U.S.A.

References

O'Shea T. and Smith B. B. (1987)."Understanding Physics by violating the laws of nature: Experiments with the Alternate Reality Kit". Proceedings of the Conference on Computer-Assisted Learning, (CAL'87), University of Strathclyde.

Pain H. and Bundy A. (1985). "What stories shall we tell novice Prolog programmers?", in Hawley R. (Ed.) Artificial Intelligence Programming Environments. Ellis-Horwood. Also published as Dept. of Artificial Intelligence research paper no. 269, Edinburgh University.

Pea R. (1986). "Language-independent conceptual "bugs" in novice programming" in Journal of Educational Computing Research, vol. 2, (1).

Perkinson H.J. (1984). Learning from our mistakes. Westport, Greenwood.

Schank R.C. (1973). Identification of Conceptualisations underlying natural language", in Computer Models of Thought and Language., Schank, R.C. and Colby K.M. (Eds.) Freeman, San Francisco.

Self J.A. (1988a). "The use of belief systems for student modelling". Paper given at the European Congress on Artificial Intelligence and Training, Lille.

Self J.A. (1988b). "Knowledge, belief and user-modelling". In O'Shea T. and Sgurev V. (Eds.), Artificial Intelligence III: Methodology, Systems, Applications. Elsevier Science Publications B.V. (North-Holland).

Self J.A. (1989). "The Case for Formalising Student Models, (and Intelligent Tutoring Systems generally)". Paper given at the 4th. Conference on Artificial Intelligence and Education, Amsterdam, May 1989.

Self J.A. (1990). "Computational Viewpoints" in Moyse R. and Elsom-Cook M.T. (Eds.), Knowledge Negotiation. (in Press), Paul Chapman, London.

Shapiro D.G. (1981). "Sniffer: a system that understands bugs". A.I. Memo no. 638, Massachusetts Institute of Technology, Mass.

Shortliffe E. H, (1976). Computer-based Medical Consultations: MYCIN. American Elsevier Publishers, New York.

Smith R.B. (1986). "The Alternate Reality Kit: An animated environment for creating interactive simulations". Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages, (Dallas, Texas), pp. 99-106.

Stevens A. L. and Collins A. (1980). "Multiple conceptual models of a complex system" in Snow R.E., Federico P. and Montague W.E. (Eds.) Aptitude Learning and Instruction Volume 2: Cognitive Process Analyses of Learning and Problem Solving, Lawrence Erlbaum, Hillsdale, NJ.

Stevens A. L., Collins A. and Goldin,S. (1979). "Misconceptions in Students' Understanding". Int. Jnl. of Man-Machine Studies, 1979, 11, pp. 145-156. Reprinted in Sleeman D. and Brown J.S. (Eds.) (1982) Intelligent Tutoring Systems. Academic Press, London.

Stevens A.L. and Roberts B. (1983). "Quantitative and Qualitative Simulations in Computer Based Training". Journal of Computer-Based Instruction, vol. 10, no. 1, pp. 16-19.

Stevens A. and Steinberg C. (1981). "A typology of explanations and its application to intelligent computer-aided instruction". Report no. 4626, Bolt, Beranek and Newman Inc. Cambridge, Mass. U.S.A.

Souther A., Acker L., Lester J. and Porter B. (1989). "Using view types to generate explanations in intelligent tutoring systems". Proceedings of the 11th. Annual Conference of the Cognitive Science Society, University of Michigan, Ann Arbor, Michigan, USA. pp. 123-129.

Suthers D. (1988). "Providing multiple views of reasoning for explanation". Proceedings of ITS-88, Montreal, June 1-3 1988, pp. 435-442.

Taylor J. (1988). "Programming in Prolog: an in-depth study of problems for beginners learning to program in Prolog". Cognitive Science Research Paper no. 111, School of Cognitive and Computing Sciences, University of Sussex, Sussex, UK.

Weiser M. and Lyle J. (1986). "Empirical studies of programmers", in Proceedings of the 1st. Workshop on Empirical Studies of Programmers, Soloway E. and Iyengar G. (Eds.), Ablex NJ. pp. 187-197.

Wenger E. (1987). Artificial Intelligence and Tutoring Systems. Morgan Kaufmann Publishers Inc. Los Altos, Ca.

White B.Y. and Frederiksen J.R. (1986). "Intelligent Tutoring Systems based upon qualitative model evolutions". In the Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, Pennsylvania.

Whitehead A.N. (1932). The Aims of Education and Other Essays. Williams and Northgate Ltd., London.

Woolf B.P. (1988). "Representing complex knowledge in an intelligent machine tutor" in Self J. (Ed.) Artificial Intelligence and human learning: intelligent compter-aided instruction. Chapman and Hall, London.

Young R. M. (1983). "Surrogates and Mappings: Two Kinds of Conceptual Models for Interactive Devices" in Gentner D. and Stevens D. L. (Eds.) (1983). Mental Models. Lawrence Erlbaum Associates, Hillsdale, NJ.

Appendix 1.

An example execution history from VIPER.

Query: unhappy(X)

Code:
```
unhappy(man):- huge(black, dog).
huge(red, fish).
huge(black, dog).
```

Result: unhappy(man).

Execution History.

history(1, 1, ?, 'unhappy(_186)', all, '[database]').

history(1, 2, >, 'unhappy(_186)', 1, '[unhappy(man), [58, 45], [huge(black, dog)], .]').

history(1, 3, /, 'unhappy(_186)', 1, '[unhappy(man), [58, 45], [huge(black, dog)], .]').

history(1, 4, ≥, 'unhappy(_186)', 1, '[unhappy(man), [58, 45], [huge(black, dog)], .]').

history(1, 5, u, ['_', '1', '8', '6'], [m, a, n], no_clause).

history(1, 6, 'v^', 'unhappy(_186)', 1, '[unhappy(man), [58, 45], [huge(black, dog)], .]').

history(1, 7, +**, 'unhappy(_186)', 1, '[unhappy(man), [58, 45], [huge(black, dog)], .]').

history(1, 8, '?sub', subgoal, 1, no_clause).

history(1, 9, >, 'huge(black, dog)', 1, '[unhappy(man), [58, 45], [huge(black, dog)], .]').

history(1, 10, \, 'huge(black, dog)', 1, '[unhappy(man), [58, 45], [huge(black, dog)], .]').

history(1, 11, <, 'huge(black, dog)', 1, '[unhappy(man), [58, 45], [huge(black, dog)], .]').

history(1, 12, >, 'huge(black, dog)', 2, '[[huge(red, fish)], .]').

history(1, 13, /, 'huge(black, dog)', 2, '[[huge(red, fish)], .]').

history(1, 14, ≥, 'huge(black, dog)', 2, '[[huge(red, fish)], .]').

history(1, 15, '-u', [b, l, a, c, k], [r, e, d], no_clause).

history(1, 16, '-v', 'huge(black, dog)', 2, '[[huge(red, fish)], .]').

history(1, 17, <, 'huge(black, dog)', 2, '[[huge(red, fish)], .]').

history(1, 18, >, 'huge(black, dog)', 3, '[[huge(black, dog)], .]').

history(1, 19, /, 'huge(black, dog)', 3, '[[huge(black, dog)], .]').

history(1, 20, ≥, 'huge(black, dog)', 3, '[[huge(black, dog)], .]').

history(1, 21, u, [b, l, a, c, k], [b, l, a, c, k], no_clause).

history(1, 22, u, [d, o, g], [d, o, g], no_clause).

history(1, 23, 'v^', 'huge(black, dog)', 3, '[[huge(black, dog)], .]').

history(1, 24, +*, 'huge(black, dog)', 3, '[[huge(black, dog)], .]').

history(1, 25, +, subgoal, 1, no_clause).

history(1, 26, ++, subgoal, all, no_clause).

history(1, 27, +***, 'unhappy(_186)', 1, '[unhappy(man), [58, 45], [huge(black, dog)], .]').

Appendix 2.

The documents used for VIPER's evaluation.

Appendix 2.1: VIPER evaluation briefing document.

Viewpoint-based Instruction for Prolog Error Recognition.

VIPER is the prototype of a tutoring system which is designed to incorporate multiple viewpoints on the domain of bug localisation in Prolog for beginners. The central assumption is that different viewpoints will help to localise different bugs in the code. The purpose of the system is to tutor the skill of applying the different viewpoints in a simplified Prolog environment, so as to localise different bugs.

- **The purposes of this evaluation are:**
  - To check that the components of the system function properly together; (ie. does it run?).
  - To test the usefulness of the way in which the different viewpoints have been encoded in the system; (ie. can the system exploit them usefully?).
  - To test the overall design of the system; (ie. can it carry out any useful tutoring, and what are its limitations?).
  - To assess the usefulness of the viewpoints to Prolog beginners; (ie. are these useful irrespective of the system's effectiveness?)

- **The evaluation has four parts.**
  - Three different viewpoints on Prolog execution will be presented on paper so that those taking part can become familiar with them.
  - Using the system, the participants will apply the viewpoints to describe the execution of a specific 'Prolog' query and database. If required, the system can provide demonstration, explanation, and corrections during this exercise.
  - When the participants can confidently apply the viewpoints to straightforward Prolog execution, they will be asked to apply the models to solve some simple debugging problems.
  - When the debugging exercises are complete, the participants will be asked to complete a short questionnaire about the system.

The Viewpoints.

The viewpoints encoded in the system describe a simplified version of Prolog execution. This does not include backtracking or the use of the 'cut'. Thus when a goal or subgoal has tried all available clauses without success, it, and its parent goal, immediately fail completely, and there is no attempt to 're-do' any goals.

The Debugging exercises.

- The system will describe a query, a simple Prolog database, and the result obtained by running the query with the database. The result always involves the binding of a variable in the query to a particular value. The system will also describe a different result, where the variable is bound to a different value. This different result is taken to be the correct one. This implies that the database of clauses shown contains a bug. A number of different bugs are associated with each of the viewpoints on Prolog execution which were explored in the first part of the evaluation, and it is one of these which produces the incorrect result.

- The correct result is produced by an ideal version of the code, and it is important to note that there is only **one** difference allowed between the bugged and the ideal code, that of the bug which gives the different result.

- The participants are asked to:
    - Indicate which clause contains the bug. They may ask questions about the ideal code via a menu.
    - When the correct clause has been selected, the participants are asked to describe the execution of the bugged clause with the relevant goal in terms of the three viewpoints on Prolog execution rehearsed earlier. Finally, they are asked to state which bug from the available range is present in the visible code. If required, the system can provide an explanation of the current bug.

Appendix 2.2: Models used in VIPER's evaluation for the description of a subset of Prolog Execution.

The student describes the execution of a simplified version of Prolog through menu selections in terms of the three models given below. The system can demonstrate this skill.

Search Strategy.

1. If there is a goal, try to resolve it with the head of the first/next database item.
2. If the head resolves consider subgoals.
3. If there are untried subgoals set the first as a goal with the full Search Space.
4. If there are no subgoals, or none left untried, the goal from the head resolution succeeds.
5. If a subgoal fails, or the head resolution fails, the whole resolution fails.
6. If a resolution fails, try to resolve the goal with the next item in the database.

Resolution.

1. Check the functors. If they unify, continue
2. If the functors do not unify, fail.
3. If the functors unify, check the arity.
4. If the arity is different, fail.
5. If the arity is the same, attempt to unify the arguments.
6. Untried arguments unify if they are one of the following:
    a) identical constants.
    b) A variable bound to the value of the other term, (a constant).
    c) An unbound variable and a constant.
    d) 2 unbound variables.
    e) 2 variables bound to the same value.
7. If any pair of arguments do not unify, fail.
8. If there are no arguments, or none left untried, return to search strategy.
9. If a goal contains an operator, and can be evaluated as 'true' with the current variable bindings, the goal succeeds.
10. If a goal contains an operator, and cannot be evaluated as 'true' with the current variable bindings, the goal fails.

Search Space.

1.A goal and generated subgoals must be proved within the same Search Space.

2. If all Search Space clauses are tried without success, a goal fails.

Appendix 2.3: The menu options relating to each model.

Search Strategy.

- Try goal with next clause
- Quit this resolution
- Subgoal Ok: try next subgoal
- All subgoals Ok, parent Ok
- Subgoal fails: parent fails
- Proved on fact, no subgs.
- Resolves with head, try subgs.
- Search complete

Resolution.

- Functors Ok.
- Functors fail
- Arity Ok.
- Arity fails
- Argument pair unify
- Argument pair fail
- All arguments unified
- Goal with operator Ok.
- Goal with operator fails.

Search Space.

- Prove new goal with S.Space
- Prove subgoal with S.Space
- Fail: whole S.Space tried

If an incorrect choice is made, the system will immediately correct it. If required, the system will demonstrate the application of the relevant rule to the current step of the execution.

Appendix 2.4: The queries and program databases used in Dialogue 1.

Query	Database
big(X)	big(X):- hairy(X). hairy(X).
unhappy(Person)	unhappy(man):- big(X). big(dog). huge(red, fish). huge(black, dog).
unhappy(Person)	big(dog) unhappy(Y):- big(Y), green(Y). unhappy(Y):- blue(Y). green(fish). green(dog). blue(alien).
unhappy(X)	big(dog) unhappy(Y):- big(Y), green(Y). unhappy(Y):- blue(Y). green(fish). green(dog). blue(alien).
gparent(pete, paul)	gparent(X, Y):- parent(X, Z), parent(Z, Y). parent(pete, helen). parent(helen, paul).

Appendix 2.5

The possible bugs associated with each viewpoint.

Resolution:

Wrong functor.

Wrong arity.

Wrong argument.

Wrong Operator.

Search Space.

Missing clause.

Extra clause.

Wrong subgoal.

Missing subgoal.

Extra Subgoal.

Search Strategy.

Wrong clause order.

Wrong subgoal order.

Appendix 3.

The queries and code used for Dialogues 2 and 3 of the evaluation.

Resolution Viewpoint.

Query	Ideal Program Database	Bugged Program Database
huge(X, Y)	big(dog). unhappy(man):- huge(black, dog). huge(red, fish). huge(black, dog).	big(dog). unhappy(man):- huge(black, dog). big(red, fish). huge(black, dog).
huge(X, Y)	big(dog). unhappy(man):- huge(black, dog). big(red, fish). huge(black, dog).	big(dog). unhappy(man):- huge(black, dog). huge(red, fish). huge(black, dog).
precious(X,Y))	big(dog). unhappy(man):- huge(black, dog). precious(red, bird). precious(black, cat).	big(dog). unhappy(man):- huge(black, dog). precious(tasty, red, bird). precious(black, cat).
huge(X, Y)	big(dog). unhappy(man):- huge(black, dog). huge(tasty, red, fish). huge(black, dog).	big(dog). unhappy(man):- huge(black, dog). huge(red, fish). huge(black, dog).
wild(red, D)	big(dog). unhappy(man):- huge(black, dog). wild(red, fish). wild(red, beastie).	big(dog). unhappy(man):- huge(black, dog). wild(green, fish). wild(red, beastie).
huge(red, D)	big(dog). unhappy(man):- huge(black, dog). huge(fish, red). huge(red, dog).	big(dog). unhappy(man):- huge(black, dog). huge(red, fish). huge(red, dog).

Search Strategy Viewpoint.

Query	Ideal Program Database	Bugged Program Database
unhappy(X)	big(dog). unhappy(Y):- green(Y), big(Y). unhappy(Y):- blue(Y). green(fish). green(dog). blue(alien).	big(dog). unhappy(Y):- big(Y), green(Y). unhappy(Y):- blue(Y). green(fish). green(dog). blue(alien).
heavy(red, D)	big(truck). heavy(red, truck). unhappy(man):- heavy(black, truck). heavy(red, train).	big(truck). unhappy(man):- heavy(black, truck). heavy(red, train). heavy(red, truck).
huge(red, D)	big(dog). huge(red, fish). unhappy(man):- huge(black, dog). huge(red, dog).	huge(red, dog). big(dog). unhappy(man):- huge(black, dog). huge(red, fish).
unhappy(X)	big(dog). unhappy(Y):- big(Y), green(Y). unhappy(Y):- blue(Y). green(fish). green(dog). blue(alien).	big(dog). unhappy(Y):- green(Y), big(Y). unhappy(Y):- blue(Y). green(fish). green(dog). blue(alien).

Search Space Viewpoint.

Query	Ideal Program Database	Bugged Program Database
height(F, D)	big(cloud). height(cloud, low). unhappy(man):- height(cloud, low). height(mountain, high).	big(cloud). unhappy(man):- height(cloud, low). height(mountain, high).
huge(F, D)	big(dog). unhappy(man):- huge(black, dog). huge(black, dog).	big(dog). huge(red, fish). unhappy(man):- huge(black, dog). huge(black, dog).
huge(F, D)	big(dog). huge(red, fish). huge(black, dog). unhappy(man):- huge(black, dog).	big(dog). huge(black, dog). unhappy(man):- huge(black, dog).
unhappy(X)	big(dog). unhappy(Y):- big(Y). unhappy(Y):- blue(Y). huge(fish). blue(alien).	big(dog). unhappy(Y):- big(Y), green(Y). unhappy(Y):- blue(Y). huge(fish). blue(alien).
unhappy(X)	big(dog). unhappy(Y):- big(Y), green(Y). unhappy(Y):- blue(Y). green(fish). green(dog). huge(fish). blue(alien).	big(dog). unhappy(Y):- huge(Y), green(Y). unhappy(Y):- blue(Y). green(fish). green(dog). huge(fish). blue(alien).
unhappy(X)	big(dog). unhappy(Y):- big(Y), green(Y). unhappy(Y):- blue(Y). huge(fish). blue(alien).	big(dog). unhappy(Y):- big(Y). unhappy(Y):- blue(Y). huge(fish). blue(alien).

Appendix 4.

The Evaluation Questionnaire.

Section 1. Experience of Prolog.

- 1.1 How long have you been learning Prolog?
- 1.2 How would you rate your ability in Prolog? (poor, fair, middling, good, very good).
- 1.3 Did you have a clear model of Prolog execution before this session?

Section 2. The Interface.

- 2.1 Did any parts of the interface not function correctly?
- 2.2 Did you find any parts of the interface difficult to use? (buttons, menus, etc.).
- 2.3 Did you find that any part of the interface was particularly useful?
- 2.4 Is there anything that you think should be added to the interface?

Section 3. The Viewpoint Representations

- 3.1 Please comment on the way in which the system described execution, (ie. in terms of three viewpoints, each composed of a set of rules).
- 3.2 Please comment on the way that the system used the different rule-parts; (ie. to describe execution, to take input from the user, and to describe the effects of bugs).
- 3.3 Please comment on the way the system related different viewpoints to different categories of bugs.
- 3.4 Please comment on the exercise which asked you to identify the bugged clause, and the information available to you at this point.
- 3.5 Were the exercises of Part 2, (describing the execution of the bugged code and identifying the bug), useful in relating viewpoints to categories of bugs? (Please explain).

Section 4. The system.

• 4.1 The system you have just used does not yet have any diagnostic mechanisms built into it, and can thus only adapt to a user in very limited ways. Please assume that such mechanisms could be added, and comment on the usefulness of the resulting system in relation to Prolog novices.

• 4.2 The system can focus its activity on a particular viewpoint with which a student is having difficulty or which interests them, (eg. Search Strategy or Resolution). Please comment on the usefulness of this feature.

• 4.3 Did working with the system add anything to your understanding of Prolog execution? Please explain.

Section 5. The Viewpoints.

Please answer these questions without reference to the specific system you have just used.

• 5.1 Are these viewpoints on Prolog useful? (Please explain)

• 5.2 Can you think of any other viewpoints which would be useful?